APPENDIX I

APPLICATION FOR

UNITED STATES LETTERS PATENT

TITLE: GRAPHICAL FUNCTIONS

APPLICANT: VIJAYA RAGHAVAN AND JAY RYAN TORGERSON

PAGES: 143

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL298430458US

I hereby certify under 37 CFR §1.10 that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

May 14, 2001
Date of Deposit

Samantha Bell
Signature

Samantha Bell
Typed or Printed Name of Person Signing Certificate

Overview

What Is Stateflow?

Stateflow is a powerful graphical design and development tool for complex control and supervisory logic problems. Using Stateflow you can:

- Visually model and simulate complex reactive systems based on *finite state machine* theory.
- Design and develop deterministic, supervisory control systems.
- Easily modify your design, evaluate the results, and verify the system's behavior at any stage of your design.
- Automatically generate integer or floating-point code directly from your design (requires Stateflow Coder).
- Take advantage of the integration with the MATLAB and Simulink environments to model, simulate, and analyze your system.

Stateflow allows you to use flow diagram notation and state transition notation seamlessly in the same Stateflow diagram. Flow diagram notation is essentially logic represented without the use of states. In some cases, using flow diagram notation is a closer representation of the system's logic and avoids the use of unnecessary states. Flow diagram notation is an effective way to represent common code structures like `for` loops and `if-then-else` constructs.

Stateflow also provides clear, concise descriptions of complex system behavior using finite state machine theory, flow diagram notations, and state-transition diagrams. Stateflow brings system specification and design closer together. It is easy to create designs, consider various scenarios, and iterate until the Stateflow diagram models the desired behavior.

Examples of Stateflow Applications

A few of the types of applications that benefit from using the capabilities of Stateflow are:

- Embedded systems
  - Avionics (planes)
  - Automotive (cars)
  - Telecommunications (e.g., routing algorithms)

  - Commercial (computer peripherals, appliances, etc.)
  - Programmable logic controllers (PLCs) (process control)
  - Industrial (machinery)
- Man-machine interface (MMI)
  - Graphical user interface (GUI)
- Hybrid systems
  - Air traffic control systems (digital signal processing (DSP) + Control + MMI)

Stateflow Components

Stateflow consists of these primary components:

- Stateflow graphics editor (see Chapter 3, "Creating Charts")
- Stateflow Explorer (see Chapter 6, "Exploring and Searching Charts")
- Stateflow simulation code generator (see Chapter 9, "Building Targets")
- Stateflow Debugger (see Chapter 10, "Debugging")

Stateflow Coder is a separately available product and generates code for nonsimulation targets. (See Chapter 9, "Building Targets" for information relevant to Stateflow Coder.)

Stateflow Dynamic Checker supports run-time checking for conditions such as cyclic behavior and data range violations. The Dynamic Checker is currently available if you have a Stateflow license.

Design Approaches

Stateflow is used together with Simulink and optionally with the Real-Time Workshop (RTW), all running on top of MATLAB. MATLAB provides access to data, high-level programming, and visualization tools. The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink. Simulink supports development of continuous-time and discrete-time dynamic systems in a graphical block diagram environment. Stateflow diagrams are incorporated into Simulink models to enhance the new event-driven capabilities in Simulink (such as conditionally executed subsystems and event detection).

You can design a model starting with a Stateflow (control) perspective and then later build the Simulink model. You can also design a model starting from a Simulink (algorithmic) perspective and then later add Stateflow diagrams. You may have an existing Simulink model that would benefit by replacing Simulink logic blocks with Stateflow diagrams. The approach you use determines how, and in what sequence, you develop various parts of the model.

The collection of all Stateflow blocks in the Simulink model is a machine. When using Simulink together with Stateflow for simulation, Stateflow generates an S-function (MEX-file) for each Stateflow machine to support model simulation. This generated code is a simulation target and is called the `sfun` target within Stateflow.

Stateflow Coder generates integer or floating-point code based on the Stateflow machine. Real-Time Workshop generates code from the Simulink portion of the model and provides a framework for running generated Stateflow code in real-time. The code generated by Stateflow Coder is seamlessly incorporated into code generated by Real-Time Workshop. You may want to design a solution that targets code generated from both products for a specific platform. This generated code is specifically a RTW target and within Stateflow is called the `rtw` target.

Using Stateflow and Stateflow Coder you can generate code exclusively for the Stateflow machine portion of the Simulink model. This generated code is for stand-alone (nonsimulation) targets. You uniquely name this target within Stateflow.

In summary, the primary design approach options are:

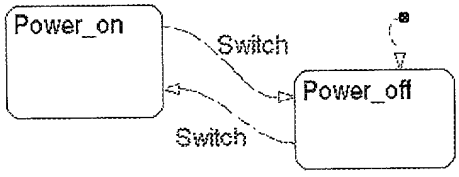
- Use Stateflow together with Simulink for simulation.
- Use Stateflow, Stateflow Coder, Simulink, and Real-Time Workshop to generate target code for the complete model.
- Use Stateflow and Stateflow Coder to generate target code for a machine.

Quick Start

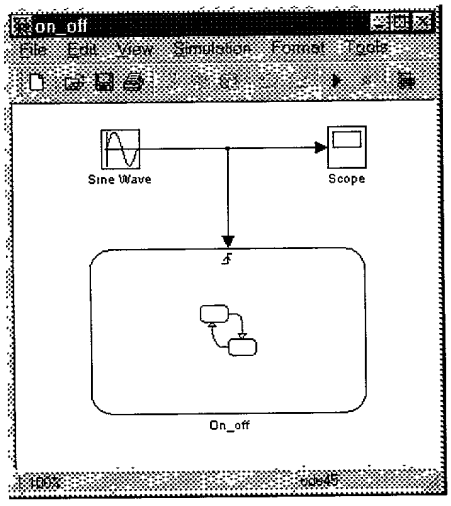
This section provides you with a quick introduction to using Stateflow. In this section, you will use Stateflow to create, run, and debug a model of a simple power switch.

The Power Switch Model

The following figure shows a Stateflow diagram that represents the power switch we intend to model.



Here is a sample of the completed Simulink model.



When you simulate this model, the generation of the input event from Simulink, `Switch`, will toggle the activity of the states between `Power_on` and `Power_off`.

Creating a Simulink Model

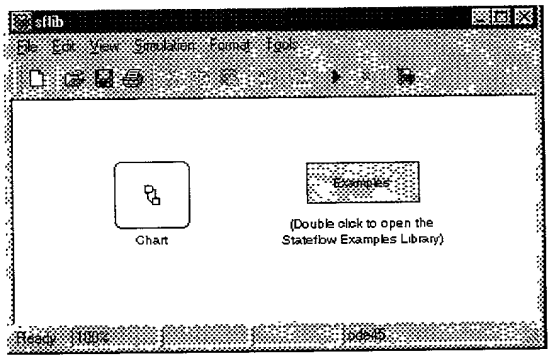
Opening the Stateflow model window is the first step toward creating a Simulink model with a Stateflow block. By default, an untitled Simulink model with an untitled, empty Stateflow block is created for you when you open the Stateflow model window. You can either start with the default empty model or copy the untitled Stateflow block into any Simulink model to include a Stateflow diagram in an existing Simulink model.

These steps describe how to create a Simulink model with a Stateflow block, label the Stateflow block, and save the model:

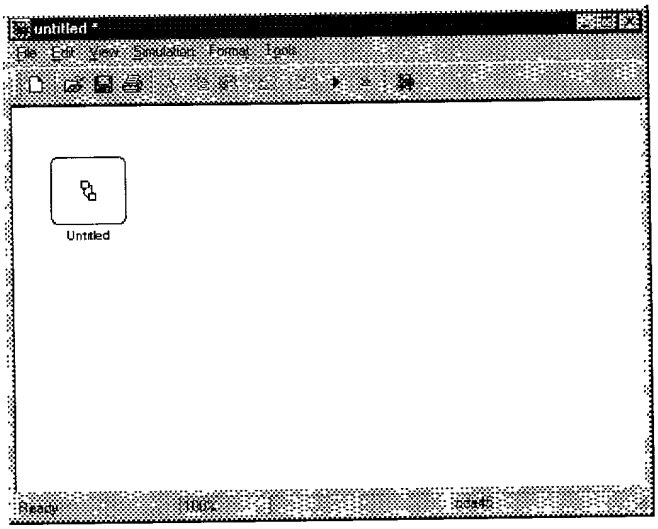
1 Display the Stateflow model window.

At the MATLAB prompt enter `stateflow`.

MATLAB displays the Stateflow block library.

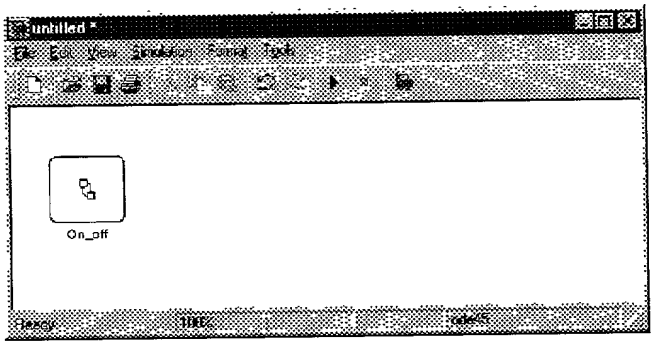


The library contains an untitled Stateflow block icon , an Examples block, and a manual switch. Stateflow also displays an untitled Simulink model window with an untitled Stateflow block.



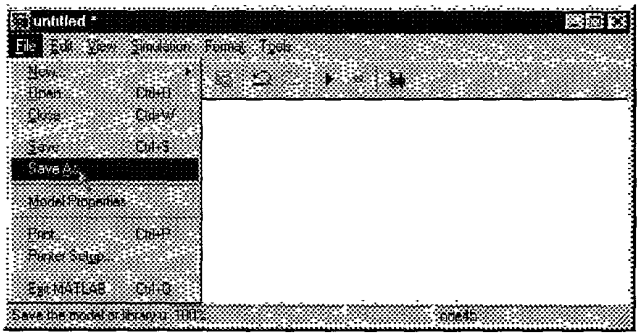
2 Label the Stateflow block.

Label the Stateflow block in the new untitled model by clicking in the text area and replacing the text "Untitled" with the text `On_off`.



3 Save the model.

Choose **Save As** from the **File** menu of the Simulink model window. Enter a model title.



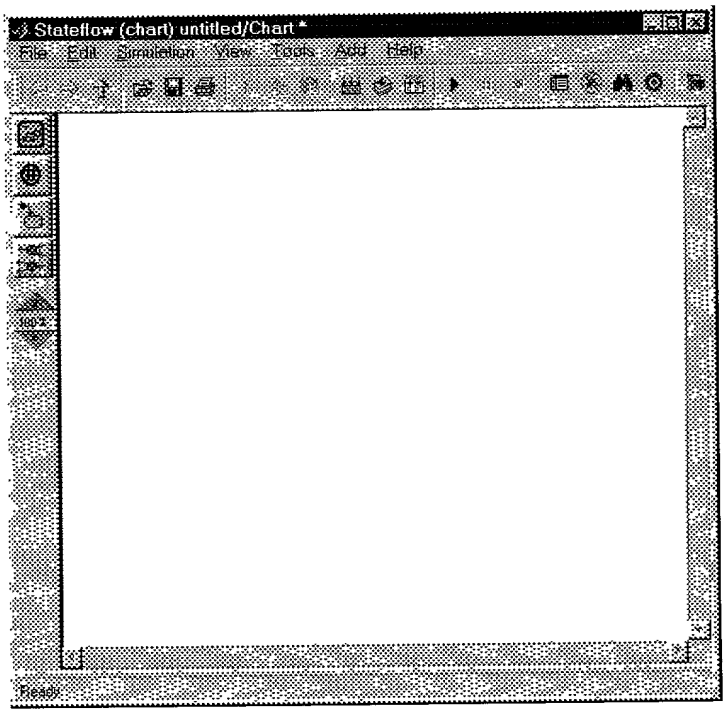
You can also save the model by choosing **Save** or **Save As** from the Stateflow graphics editor **File** menu. Saving the model either from Simulink or from the graphics editor saves all contents of the Simulink model.

Creating a Stateflow Diagram

These steps describe how to create a simple Stateflow diagram using the graphics editor:

1 Invoke the graphics editor.

Double-click on the Stateflow block in the Simulink model window to invoke the graphics editor window.

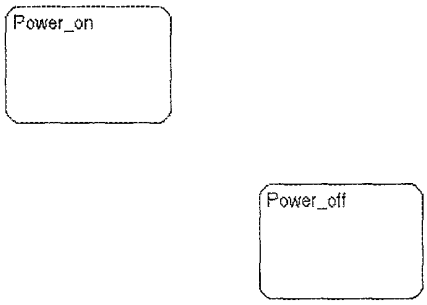


2 Create states.

Click on the **State** button  in the toolbar. Click in the drawing area to place the state in the drawing area. Position the cursor over that state, click the right mouse button, and drag to make a copy of the state. Release the right mouse button to drop the state at that location.

3 Label states.

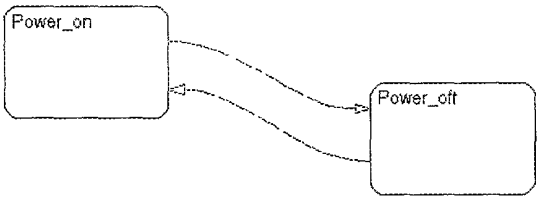
Click on the ? character within each state to enter each state label. Label the states with the titles Power_on and Power_off. Deselect the state to exit the edit. To deselect a state, click anywhere outside the state or press the **Esc** key. Your Stateflow diagram should look similar to this sample.



4 Create transitions.

Draw a transition starting from Power_on and ending at Power_off. Place the cursor at a straight portion of the border of the Power_on state. Click the border when the cursor changes to a crosshair. Without releasing the mouse button, drag the mouse to a straight portion on the border of the Power_off state. When the transition snaps to the border of the Power_off state, release the mouse button. (The crosshair will not appear if you place the cursor on a corner, since corners are used for resizing.)

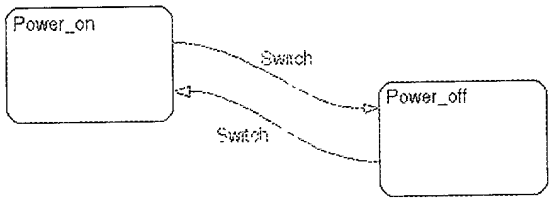
Draw another transition starting from Power_off and ending on Power_on. Your Stateflow diagram should look similar to this sample.



5 Label the transitions.

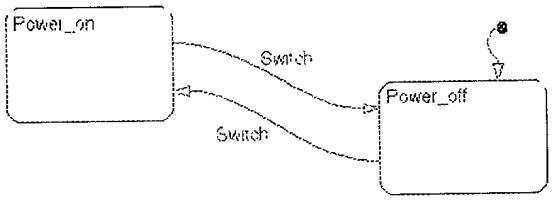
Click on the transition from Power_on to Power_off to select it. Click on the ? alongside the transition and enter the label Switch. Press the **Escape** key to deselect the transition label and exit the edit.

Label the transition from Power_off to Power_on with the same text, Switch. Your Stateflow diagram should look similar to this sample.



6 Add a default transition.

Click and release the mouse on the **Default Transition** button  in the toolbar. Drag the mouse to a straight portion on the border of the Power_off state. Click and release the mouse when the arrowhead snaps to the border of the Power_off state. Your Stateflow diagram should look similar to this sample.



For More Information

For more information on creating Stateflow diagrams using the graphics editor see Chapter 3, "Creating Charts."

Defining Input Events

Add and define input events within the Stateflow diagram:

1 Choose **Explore** from the graphics editor **Tools** menu to invoke the Explorer.

2 Double-click on the machine name (same as the Simulink model name) in the **Object Hierarchy** list.

3 Click on the `On_off` chart entry in the **Object Hierarchy** list.

4 Select **Event** from the **Add** menu.

5 Double-click the event icon in the Explorer entry for the event to display the event's property dialog.

6 Enter `Switch` in the **Name** field of the **Event properties** dialog box.

7 Select **Input from Simulink** as the **Scope** value.

8 Select **Rising Edge** as the **Trigger** type.

9 Click on the **OK** button to apply the changes and close the window.

10 Choose **Close** from the Explorer **File** menu to close the Explorer.

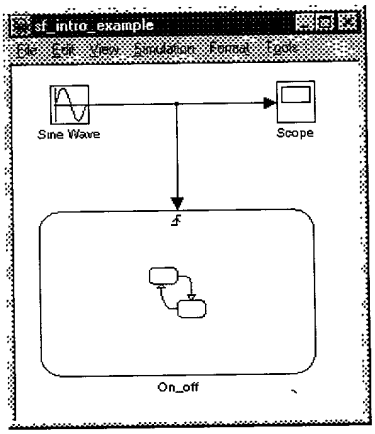
Defining the Stateflow Interface

Make connections in the Simulink model between other blocks and the Stateflow block:

1 Enter `simulink` in the MATLAB command window to invoke Simulink.

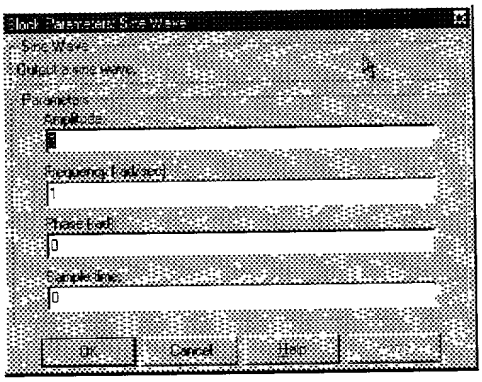
2 Add a Sine Wave block (located in the Simulink Sources block library) and connect it to the input trigger port of the Stateflow block.

3 Add a Scope block (located in the Simulink Sinks block library) and connect it to the Sine Wave block output as well. Your model should look similar to this.



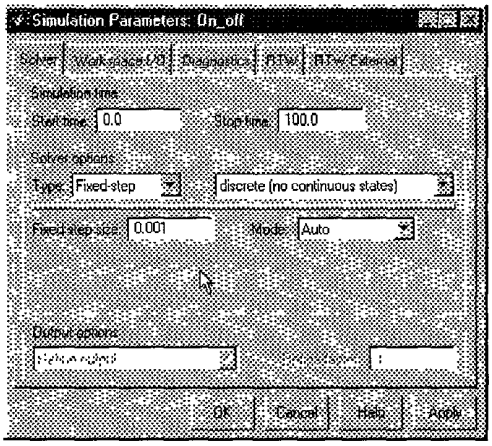
Defining Simulink Parameters

1 Double-click on the Sine Wave block and edit the parameters as shown in this example dialog box.



Click on the **OK** button to apply the changes and close the dialog box.

2 Choose **Parameters** from the **Simulation** menu of the Simulink model window and edit the values to match the values in this dialog box.



For More Information

See Chapter 5, "Defining Stateflow Interfaces."

Parsing the Stateflow Diagram

Parsing the Stateflow diagram ensures that the notations you have specified are valid and correct. To parse the Stateflow diagram, choose **Parse Diagram** from the **Tools** menu of the graphics editor. Informational messages are displayed in the MATLAB command window. Any error messages are displayed in red. If no red error messages appear, the parse operation is successful and the text `Done` is displayed.

For More Information

See "How Stateflow Builds Targets" on page 9-3.

Running a Simulation

Note Running a simulation may require setting up the tools used to build Stateflow targets. See "Setting Up Target Build Tools" on page 9-5 for more information.

These steps show how to run a simulation:

1 Ensure that the Stateflow diagram and the Scope block are open.

   Double-click on the *On_off* Stateflow block to display the Stateflow diagram. Double-click on the Scope block to display the output of the Sine wave.

2 Select **Open Simulation Target** from the graphics editor **Tools** menu.

   The **Simulation Target Builder** dialog box appears.

3 Select **Coder Options** on the **Simulation Target Builder** dialog box.

   The **Simulation Coder Options** dialog box appears.

4 Ensure that the check box to **Enable Debugging/Animation** is checked. Click on the **OK** button to apply the change. Close the **Simulation Coder Options** and the **Simulation Target Builder** dialog boxes.

5 Select **Debug** from the graphics editor **Tools** menu. Ensure that the **Enabled** radio button under **Animation** is checked to enable Stateflow diagram animation. Click on the **Close** button to apply the change and close the window.

6 Choose **Start** from the graphics editor **Simulation** menu to start a simulation of the model.

   By default the S-function is the simulation target for any Stateflow blocks. Stateflow displays code generation status messages in the MATLAB command window. Before starting the simulation, Stateflow temporarily sets the model to read-only to prevent accidental modification while the simulation is running.

The input from the Sine block is defined as the **Input from Simulink** event `Switch`. When the simulation starts the Stateflow diagram is animated reflecting the state changes triggered by the input sine wave. Each input event of `Switch` toggles the model between the `Power_off` and `Power_on` state.

7 Choose **Stop** from the graphics editor **Simulation** menu to stop a simulation. Once the simulation stops, Stateflow resets the model to writable.

Note Before generating code, Stateflow creates a directory called `sfprj` in the current directory if the directory does not already exist. Stateflow uses the `sfprj` directory during code generation to store information required for incremental code generation.

Debugging

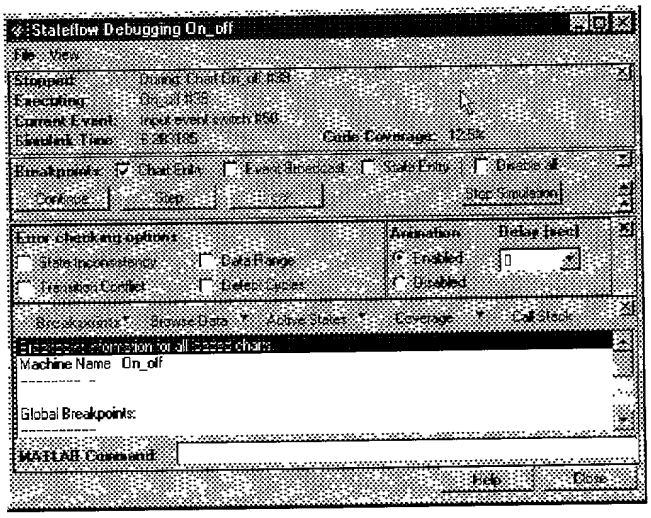
The Stateflow Debugger supports functions like single stepping, animating, and running up to a designated breakpoint and then stopping.

These steps show how to step through the simulation using the Debugger:

1 Display the Debugger by choosing **Debug** from the **Tools** menu of the graphics editor.

2 Click on the **Breakpoints**: **Chart Entry** check box to specify you want the Debugger to stop the simulation execution when the chart is entered.

3 Click on the **Start** button to start the simulation. Informational and error messages related to the S-function code generation for Stateflow blocks are displayed in the MATLAB command window. When the target is built, the graphics editor becomes read-only (frozen) and the Debugger window will be updated and look similar to this.



4 Click on the **Step** button to proceed one step at a time through the simulation. The Debugger window displays the following information:

- Where the simulation is stopped
- What is executing
- The current event
- The simulation time
- The current code coverage percentage

Watch the graphics editor window as you click on the **Step** button to see each transition and state animated when it is executing. After both `Power_off` and `Power_on` have become active by stepping through the simulation, the code coverage indicates 100%.

5 Choose **Stop** from the graphics editor **Simulation** menu to stop a simulation. Once the simulation stops, the model becomes editable.

6 Click on the **Close** button in the Debugger window.

7 Choose **Close** from the **File** menu in the Simulink model window.

For More Information

See Chapter 10, "Debugging" for more information beyond the debugging topics in this section.

Generating Code

When you simulate a Simulink model containing Stateflow charts, Stateflow generates a Simulink S-function (`sfun` target) that enables Simulink to simulate the Stateflow blocks. The `sfun` target can be used only with Simulink. If you have the Stateflow Coder, you can generate stand-alone code suitable for a particular processor. See Chapter 9, "Building Targets" for more information on code generation.

2

How Stateflow Works

Finite State Machine Concepts

What Is a Finite State Machine?

A *finite state machine* (FSM) is a representation of an event-driven (reactive) system. In an event-driven system, the system transitions from one state (mode) to another prescribed state, provided that the condition defining the change is true.

For example, you can use a state machine to represent a car's automatic transmission. The transmission has a number of operating states: park, neutral, drive, reverse, and so on. The system transitions from one state to another when a driver shifts the stick from one position to another, for example, from park to neutral.

FSM Representations

Traditionally, designers used truth tables to represent relationships among the inputs, outputs, and states of an FSM. The resulting table describes the logic necessary to control the behavior of the system under study. Another approach to designing event-driven systems is to model the behavior of the system by describing it in terms of transitions among states. The state that is active is determined based on the occurrence of events under certain conditions. State-transition diagrams (STDs) and bubble diagrams are graphical representations based on this approach.

Stateflow Representations

Stateflow uses a variant of the finite state machine notation established by Harel [1]. Using Stateflow, you create Stateflow diagrams. A Stateflow diagram is a graphical representation of a finite state machine where *states* and *transitions* form the basic building blocks of the system. You can also represent flow (stateless) diagrams using Stateflow. Stateflow provides a block that you include in a Simulink model. The collection of Stateflow blocks in a Simulink model is the Stateflow machine.

Additionally, Stateflow enables the representation of hierarchy, parallelism, and history. Hierarchy enables you to organize complex systems by defining a parent/offspring object structure. For example, you can organize states within other higher-level states. A system with parallelism can have two or more orthogonal states active at the same time. History provides the means to

specify the destination state of a transition based on historical information. These characteristics enhance the usefulness of this approach and go beyond what STDs and bubble diagrams provide.

Notations

A notation defines a set of objects and the rules that govern the relationships between those objects. Stateflow notation provides a common language to communicate the design information conveyed by a Stateflow diagram.

Stateflow notation consists of:

- A set of graphical objects
- A set of nongraphical text-based objects
- Defined relationships between those objects

See Chapter 7, "Notations," for detailed information on Stateflow notations.

Semantics

Semantics describe how the notation is interpreted and implemented. A completed Stateflow diagram illustrates how the system will behave. A Stateflow diagram contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during Stateflow diagram execution.

Knowledge of the semantics is important to make sound Stateflow diagram design decisions for code generation. Different use of notations results in different ordering of simulation and generated code execution.

The default semantics provided with the product are described in Chapter 8, "Semantics."

References

For more information on finite state machine theory, consult these sources:

[1] Harel, David, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming 8*, 1987, pages 231-274.

[2] Hatley, Derek J. and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing Co., Inc., NY, 1988.

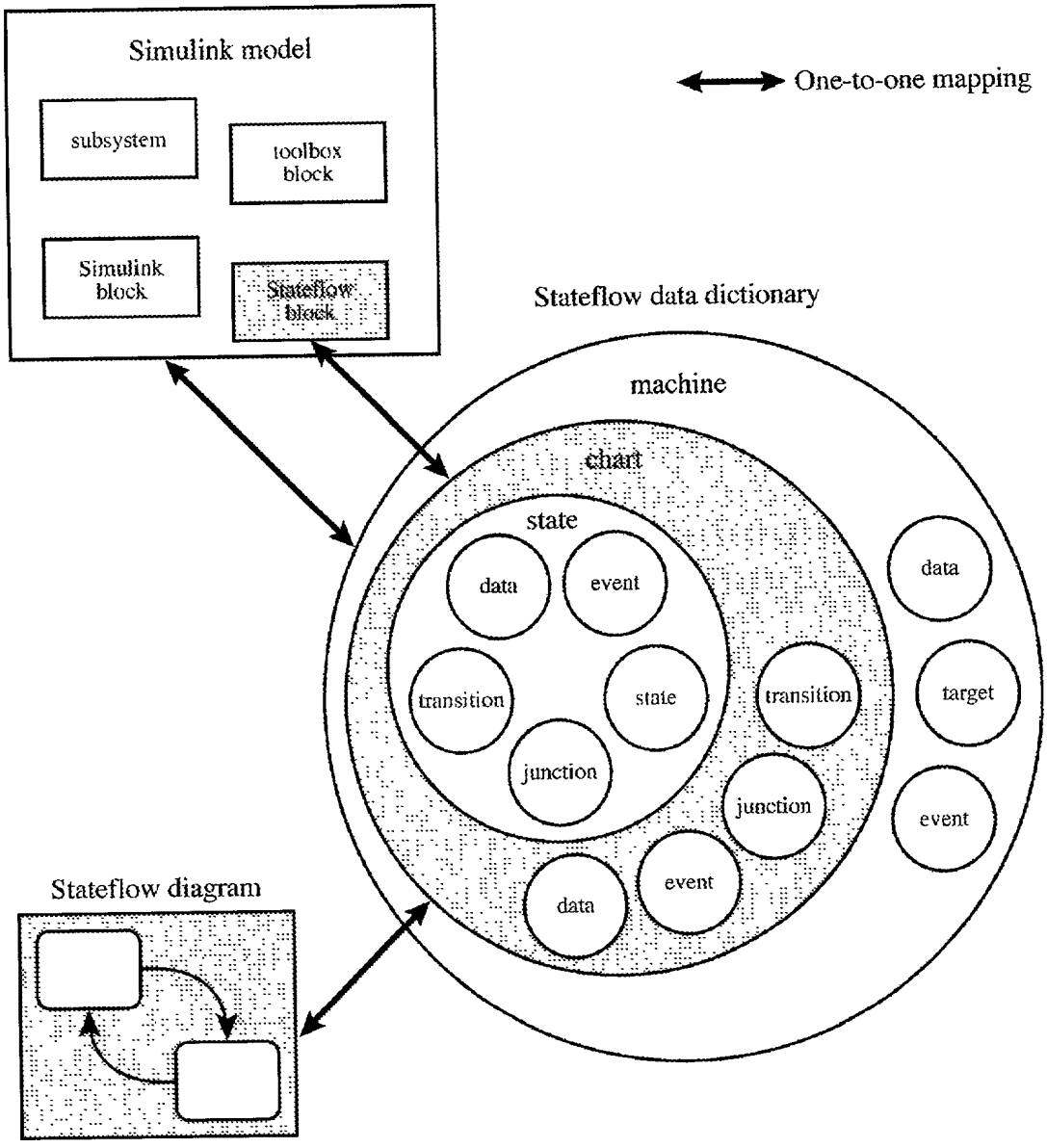
Anatomy of a Model and Machine

The Simulink Model and Stateflow Machine

The Stateflow machine is the collection of Stateflow blocks in a Simulink model. The Simulink model and Stateflow machine work seamlessly together. Running a simulation automatically executes both the Simulink and Stateflow portions of the model.

A Simulink model can consist of combinations of Simulink blocks, toolbox blocks, and Stateflow blocks (Stateflow diagrams). In Stateflow, the chart (Stateflow diagram) consists of a set of graphical (states, transitions, connective junctions, and history junctions) and nongraphical (event, data, and target) objects.

There is a one-to-one correspondence between the Simulink model and the Stateflow machine. Each Stateflow block in the Simulink model is represented in Stateflow by a single chart (Stateflow diagram). Each Stateflow machine has its own object hierarchy. The Stateflow machine is the highest level in the Stateflow hierarchy. The object hierarchy beneath the Stateflow machine consists of combinations of the graphical and nongraphical objects. The data dictionary is the repository for all Stateflow objects.


Simulink model
subsystem
toolbox block
Simulink block
Stateflow block
One-to-one mapping
Stateflow data dictionary
machine
chart
state
data
event
transition
state
junction
transition
junction
event
data
target
event
data
Stateflow diagram

Stateflow scoping rules dictate where the types of nongraphical objects can exist in the hierarchy. For example, data and events can be parented by the machine, the chart (Stateflow diagram), or by a state. Targets can only be parented by the machine. Once a parent is chosen, that object is known in the hierarchy from the parent downwards (including the parent's offspring). For example, a data object parented by the machine is accessible by that machine, by any charts within that machine, and by any states within that machine. The hierarchy of the graphical objects is easily and automatically handled for you by the graphics editor. You manage the hierarchy of nongraphical objects through the Explorer or the graphics editor **Add** menu.

Defining Stateflow Interfaces

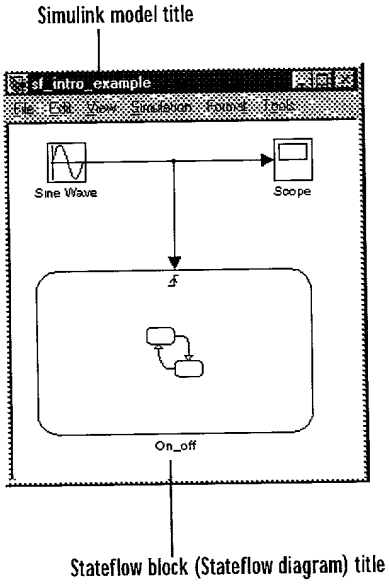
Each Stateflow block corresponds to a single Stateflow diagram. The Stateflow block interfaces to its Simulink model. The Stateflow block may interface to code sources external to the Simulink model (data, events, custom code).

Stateflow diagrams are event driven. Events can be local to the Stateflow block or can be propagated to and from Simulink and code sources external to Simulink. Data can be local to the Stateflow block or can be shared with and passed to the Simulink model and to code sources external to the Simulink model.

You must define the interface to each Stateflow block. Defining the interface for a Stateflow block can involve some or all of these tasks:

- Defining the Stateflow block update method
- Defining **Output to Simulink** events
- Adding and defining nonlocal events and nonlocal data within the Stateflow diagram
- Defining relationships with any external sources

In this example, the Simulink model titled `sf_intro_example` consists of a Simulink Sine Wave source block, a Simulink Scope sink block, and a single Stateflow block, titled `On_off`.



See "Defining Input Events" on page 4-7 and Chapter 5, "Defining Stateflow Interfaces," for more information.

Stateflow Diagram Objects

This sample Stateflow diagram highlights some key graphical components. The sections that follow describe these graphical components as well as some nongraphical objects and related concepts in greater detail.

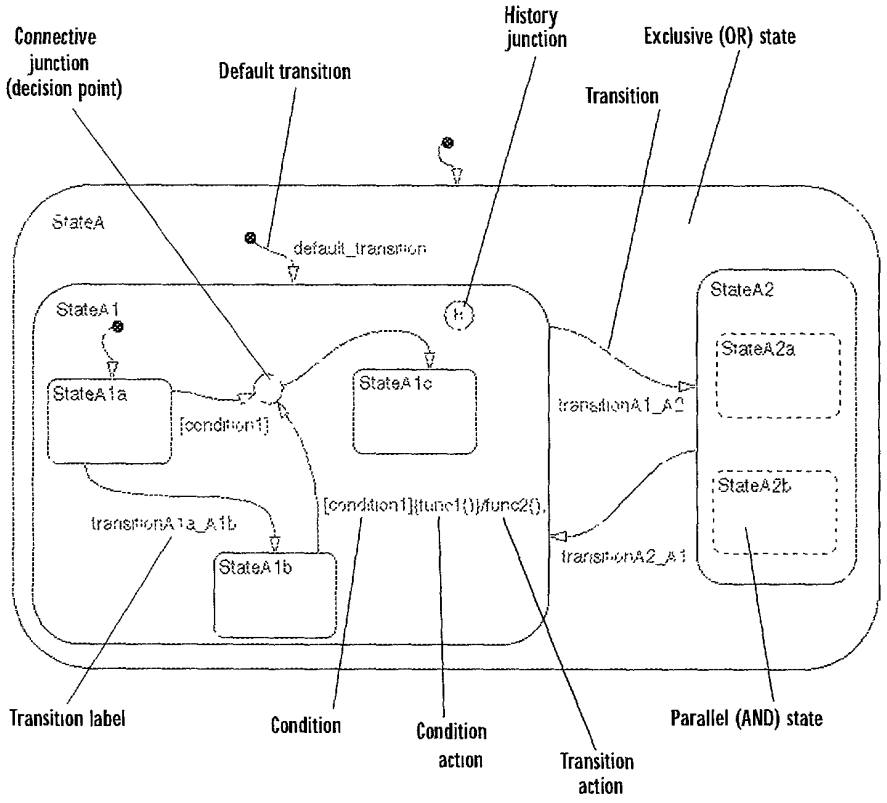


Figure 2-1: Graphical Components

States

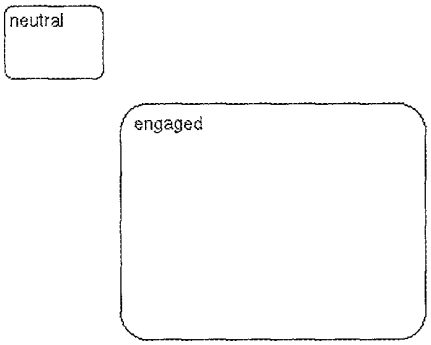
A *state* describes a mode of an event-driven system. The activity or inactivity of the states dynamically changes based on events and conditions.

Every state has a parent. In a Stateflow diagram consisting of a single state, that state's parent is the Stateflow diagram itself (also called the Stateflow diagram root). You can place states within other higher-level states. In the figure, StateA1 is a child in the hierarchy to StateA.

A state also has history. History provides an efficient means of basing future activity on past activity.

States have labels that can specify actions executed in a sequence based upon action type. The action types are `entry`, `during`, `exit`, and `on`.

In an automatic transmission example, the transmission can either be in neutral or engaged in a gear. Two states of the transmission system are `neutral` and `engaged`.

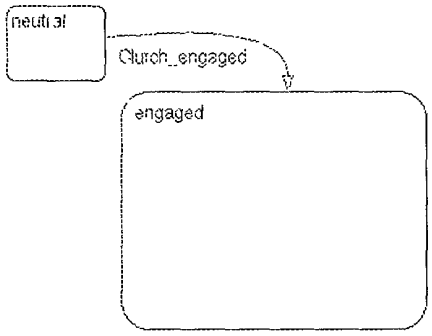


Stateflow provides two types of states: parallel (AND) and exclusive (OR) states. You represent parallelism with AND (parallel) states. The transmission example shows exclusive (OR) states. Exclusive (OR) states are used to describe modes that are mutually exclusive. The system is either in the `neutral` state or the `engaged` state at any one time.

Transitions

A *transition* is a graphical object that, in most cases, links one object to another. One end of a transition is attached to a source object and the other end to a destination object. The *source* is where the transition begins and the *destination* is where the transition ends. A *transition label* describes the circumstances under which the system moves from one state to another. It is always the occurrence of some event that causes a transition to take place. In the figure, the transition from `StateA1` to `StateA2` is labeled with the event `transitionA1_A2` that triggers the transition to occur.

Consider again the automatic transmission system. clutch_engaged is the event required to trigger the transition from neutral to engaged.



Events

Events drive the Stateflow diagram execution. Events are nongraphical objects and are thus not represented directly in the figure. All events that affect the Stateflow diagram must be defined. The occurrence of an event causes the status of the states in the Stateflow diagram to be evaluated. The broadcast of an event can trigger a transition to occur or can trigger an action to be executed. Events are broadcast in a top-down manner starting from the event's parent in the hierarchy.

Events are created and modified using the Stateflow Explorer. Events can be created at any level in the hierarchy. Events have properties such as a scope. The scope defines whether the event is:

- Local to the Stateflow diagram
- An input to the Stateflow diagram from its Simulink model
- An output from the Stateflow diagram to its Simulink model
- Exported to a (code) destination external to the Stateflow diagram and Simulink model
- Imported from a code source external to the Stateflow diagram and Simulink model

Data

Data objects are used to store numerical values for reference in the Stateflow diagram. Data objects are nongraphical objects and are thus not represented directly in the figure.

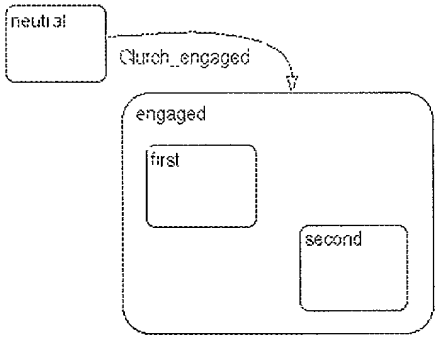
Data objects are created and modified using the Stateflow Explorer. Data objects can be created at any level in the hierarchy. Data objects have properties such as a scope. The scope defines whether the data object is:

- Local to the Stateflow diagram
- An input to the Stateflow diagram from its Simulink model
- An output from the Stateflow diagram to its Simulink model
- Non-persistent temporary data
- Defined in the MATLAB workspace
- A constant
- Exported to a (code) destination external to the Stateflow diagram and Simulink model
- Imported from a code source external to the Stateflow diagram and Simulink model

Hierarchy

Hierarchy enables you to organize complex systems by defining a parent and offspring object structure. A hierarchical design usually reduces the number of transitions and produces neat, manageable diagrams. Stateflow supports a hierarchical organization of both charts and states. Charts can exist within charts. A chart that exists in another chart is known as a subchart.

Similarly, states can exist within other states. Stateflow represents state hierarchy with superstates and substates. For example, this Stateflow diagram has a superstate that contains two substates.



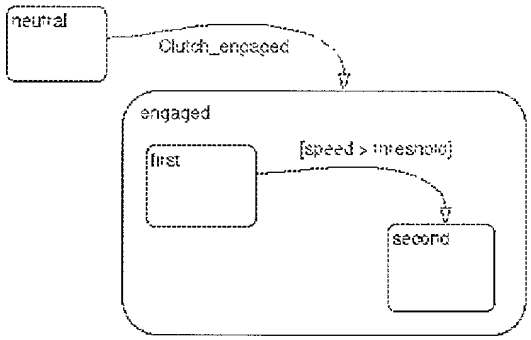
The engaged superstate contains the `first` and second substates. The engaged superstate is the parent in the hierarchy to the states `first` and second. When the event `clutch_engaged` occurs, the system transitions out of the `neutral` state to the engaged superstate. Transitions within the engaged superstate are intentionally omitted from this example for simplicity.

A transition out of a higher level, or *superstate*, also implies transitions out of any active substates of the superstate. Transitions can cross superstate boundaries to specify a substate destination. If a substate is active its parent superstate is also active.

Conditions

A *condition* is a Boolean expression specifying that a transition occurs, given that the specified expression is true. In the component summary Stateflow diagram, `[condition1]` represents a Boolean expression that must be true for the transition to occur.

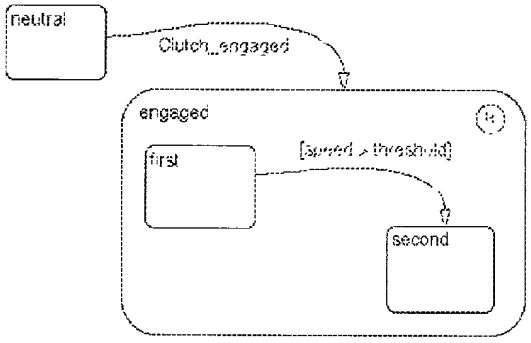
In the automatic transmission system, the transition from `first` to second occurs if the Boolean condition [`speed > threshold`] is true.



History Junction

History provides the means to specify the destination substate of a transition based on historical information. If a superstate with exclusive (OR) decomposition has a history junction, the transition to the destination substate is defined to be the substate that was most recently visited. A history junction applies to the level of the hierarchy in which it appears. The history junction overrides any default transitions. In the component summary Stateflow diagram, the history junction in `StateA1` indicates that when a transition to `StateA1` occurs, the substate that becomes active (`StateA1a`, `StateA1b`, or `StateA1c`) is based on which of those substates was most recently active.

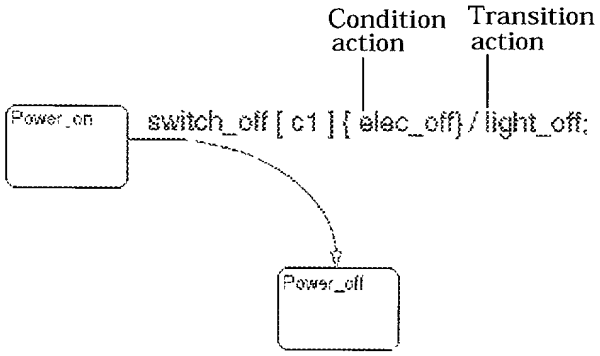
In the automatic transmission system, history indicates that when `clutch_engaged` causes a transition from `neutral` to the engaged superstate, the substate that becomes active, either `first` or second, is based on which of those substates was most recently active.



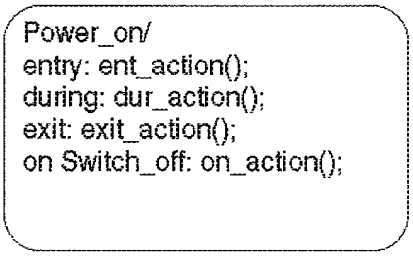
Actions

Actions take place as part of Stateflow diagram execution. The action can be executed either as part of a transition from one state to another or based on the activity status of a state. In the figure, the transition segment from StateA1b to the connective junction is labeled with a condition action (func1 ()) and a transition action (func2 ()). The semantics of how and why actions take place are discussed throughout the examples in Chapter 8, "Semantics."

Transitions can have *condition* actions and *transition* actions, as shown in this example.



States can have `entry`, `during`, `exit`, and `on` *`event_name`* actions. For example,



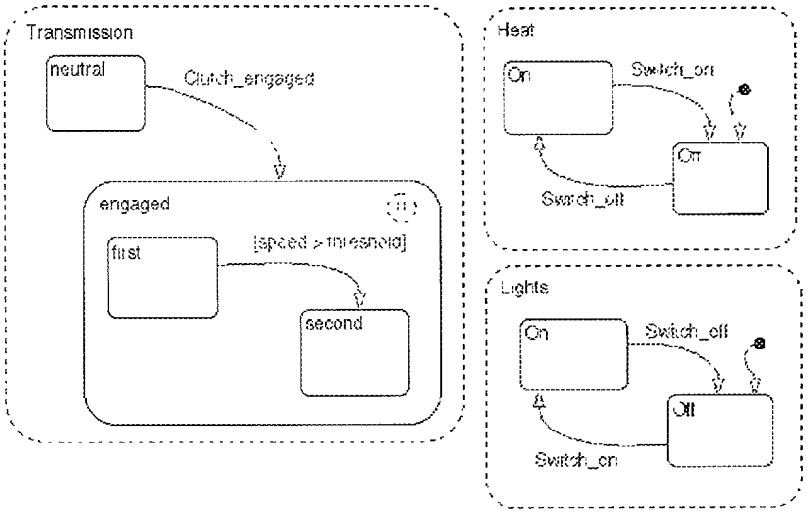
The *action language* defines the types of actions you can specify and their associated notations. An action can be a function call, an event to be broadcast, a variable to be assigned a value, etc.

Stateflow supports both Mealy and Moore finite state machine modeling paradigms. In the Mealy model, actions are associated with transitions, whereas in the Moore model they are associated with states. Stateflow supports state actions, transition actions, and condition actions. For more information, see the section titled "What Is an Action Language?" on page 7-37.

Parallelism

A system with *parallelism* has two or more states that can be active at the same time. The activity of each parallel state is essentially independent of other states. In the figure, `StateA2a` and `StateA2b` are parallel (AND) states. `StateA2` has parallel (AND) state decomposition.

For example, this Stateflow diagram has parallel superstate decomposition.



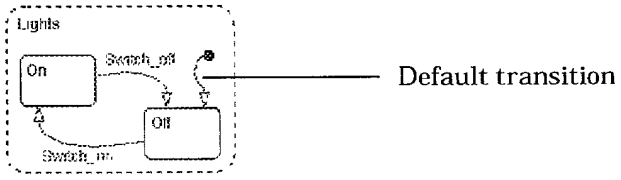
The transmission, heating, and light systems are parallel subsystems in a car. They exist in parallel and are physically independent of each other. There are many other parallel components in a car, such as the braking and windshield wiper subsystems.

You represent parallelism in Stateflow by specifying parallel (AND) state decomposition. Parallel (AND) states are displayed as dashed rectangles.

Default Transitions

Default transitions specify which exclusive (OR) state is to be active when there is ambiguity between two or more exclusive (OR) states at the same level in the hierarchy. In the figure, when `StateA` is active, by default `StateA1` is also active. Without the default transition to `StateA1`, there is ambiguity in whether `StateA1` or `StateA2` should be active.

In the Lights subsystem, the default transition to the Lights.Off substate indicates that when the Lights superstate becomes active, the Off substate becomes active by default.

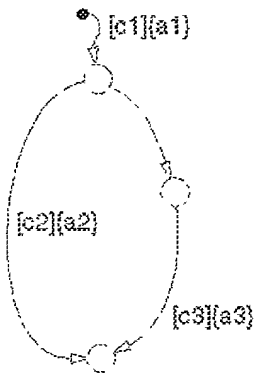


Default transitions specify which exclusive (OR) substate in a superstate the system enters by default, in the absence of any information. History junctions override default transition paths in superstates with exclusive (OR) decomposition.

Connective Junctions

Connective junctions are decision points in the system. A connective junction is a graphical object that simplifies Stateflow diagram representations and facilitates generation of efficient code. Connective junctions provide alternative ways to represent desired system behavior. In the figure, the connective junction is used as a decision point for two transition segments that complete at StateA1c.

This example shows how connective junctions (displayed as small circles) are used to represent the flow of an if code structure.




```
if [c1]{
     a1
     if [c2]{
        a2
     }else if [c3]{
       a3
     }
}
```

Exploring a Real-World Stateflow Application

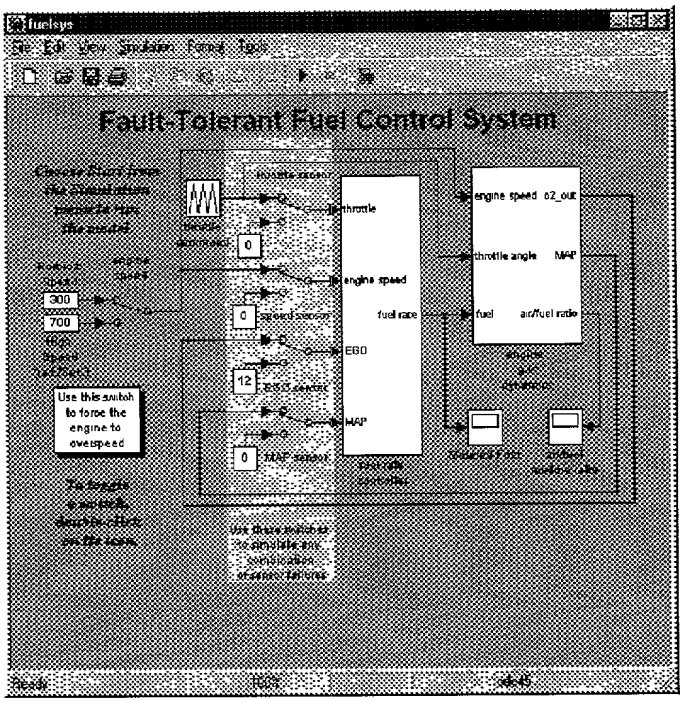
The modeling of a fault tolerant fuel control system demonstrates how Simulink and Stateflow may be used to efficiently model hybrid systems containing both continuous dynamics and complex logical behavior. Elements in the model containing time domain based dynamic behavior are modeled in Simulink, while changes in control configuration are implemented in Stateflow.

The model described represents a fuel control system for a gasoline engine. The system is highly robust in that individual sensor failures are detected and the control system is dynamically reconfigured for uninterrupted operation. This section describes how Stateflow is used to implement the supervisory logic control system dealing with the sensor failures.

Analysis and Physics

Physical and empirical relationships form the basis for the throttle and intake manifold dynamics of this model. The mass flow rate of air pumped from the intake manifold, divided by the fuel rate, which is injected at the valves, gives the air-fuel ratio. The ideal, or stoichiometric mixture ratio provides a good compromise between power, fuel economy, and emissions. A target ratio of 14.6 is assumed in this system. Typically, a sensor determines the amount of residual oxygen present in the exhaust gas (EGO). This gives a good indication of the mixture ratio and provides a feedback measurement for closed-loop control. If the sensor indicates a high oxygen level, the control law increases the fuel rate. When the sensor detects a fuel-rich mixture, corresponding to a very low level of residual oxygen, the controller decreases the fuel rate.

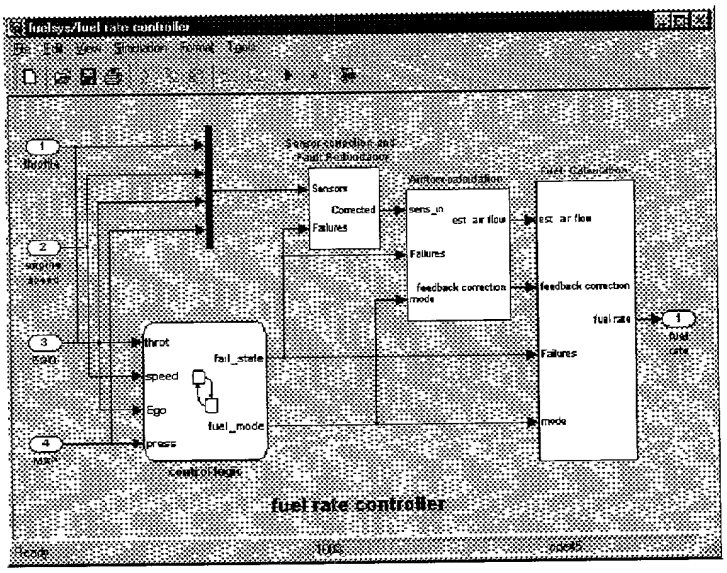
The following figure shows the top level of the Simulink model (`fuelsys.mdl`). The model is modularized into a fuel rate controller and a subsystem to simulate the engine gas dynamics.



The fuel rate controller uses signals from the system's sensors to determine the fuel rate which gives a stoichiometric mixture. The fuel rate combines with the actual air flow in the engine gas dynamics model to determine the resulting mixture ratio as sensed at the exhaust. The user can selectively disable each of the four sensors (throttle angle, speed, EGO and manifold absolute pressure [MAP]), to simulate failures. Simulink accomplishes this with Manual Switch blocks. The user can toggle the position of a switch by double-clicking its icon prior to, or during, a simulation. Similarly, the user can induce the failure condition of a high engine speed by toggling the switch on the far left. A Repeating Table block provides the throttle angle input and periodically repeats the sequence of data specified in the mask

The controller uses the sensor input and feedback signals to adjust the fuel rate to give a stoichiometric ratio. The model uses four subsystems to implement

this strategy: control logic, sensor correction, airflow calculation, and fuel calculation. Under normal operation, the model estimates the airflow rate and multiplies the estimate by the reciprocal of the desired ratio to give the fuel rate. Feedback from the oxygen sensor provides a closed-loop adjustment of the rate estimation in order to maintain the ideal mixture ratio.



A detailed explanation of the algorithmic (Simulink) part of the fault tolerant control system is given in *Using Simulink and Stateflow in Automotive Applications*, a Simulink-Stateflow Technical Examples booklet published by The MathWorks. This section concentrates on the supervisory logic part of the system that is implemented in Stateflow, but the following points are crucial to the interaction between Simulink and Stateflow:

- The supervisory logic monitors the readings from the sensors as data inputs into Stateflow.
- The logic determines from these readings which sensors have failed and outputs a failure state boolean array as `fail_state`.
- Given the current failure state, the logic determines in which fueling mode the engine should be run.

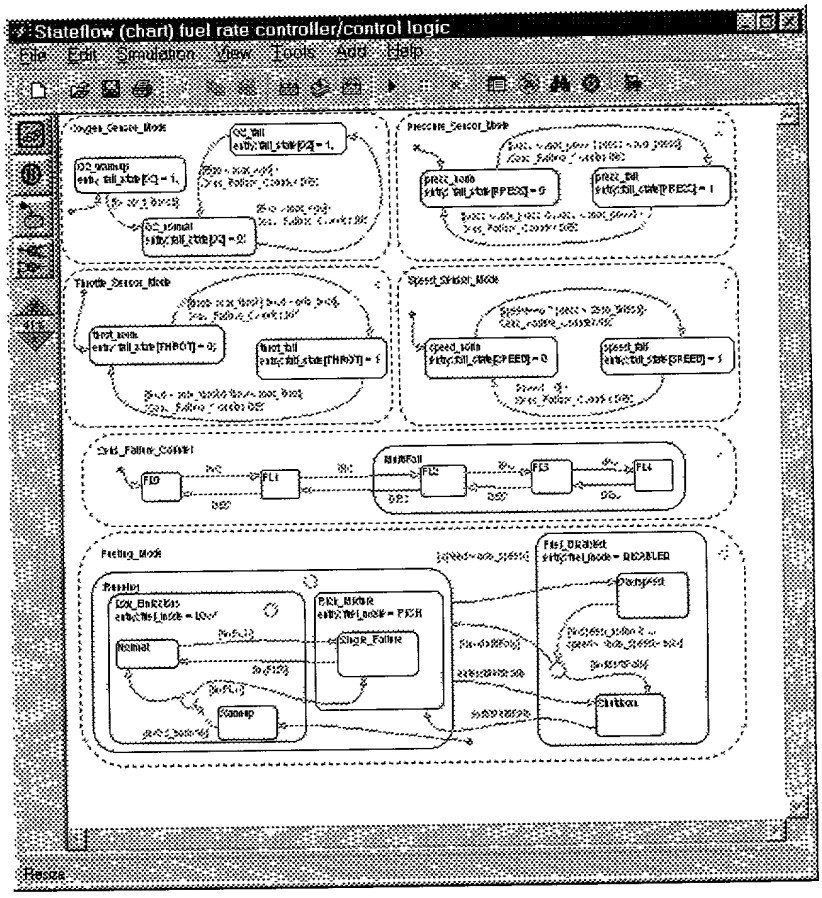
The fueling mode can be either a:

- **Low emissions mode**, the normal mode of operation where no sensors have failed
- **Rich mixture mode**, used when a sensor has failed to ensure smooth running of the engine
- **Shutdown mode**, where more than one sensor has failed rendering the engine inoperable

The fueling mode and failure state are output from the Stateflow as `fuel_mode` and `fail_state` respectively into the algorithmic part of the model where they determine the fueling calculations.

Control Logic

The single Stateflow chart that implements the entire control logic is shown below.



The chart consists of six parallel states (denoted by dash-dotted boundaries) that represent concurrent modes of operation.

The four parallel states at the top of the diagram correspond to the four individual sensors. Each state has sub-modes or sub-states that represent the status of that particular sensor, i.e., whether it has failed or not. These sub-states are mutually exclusive: if the throttle sensor has failed then it is in

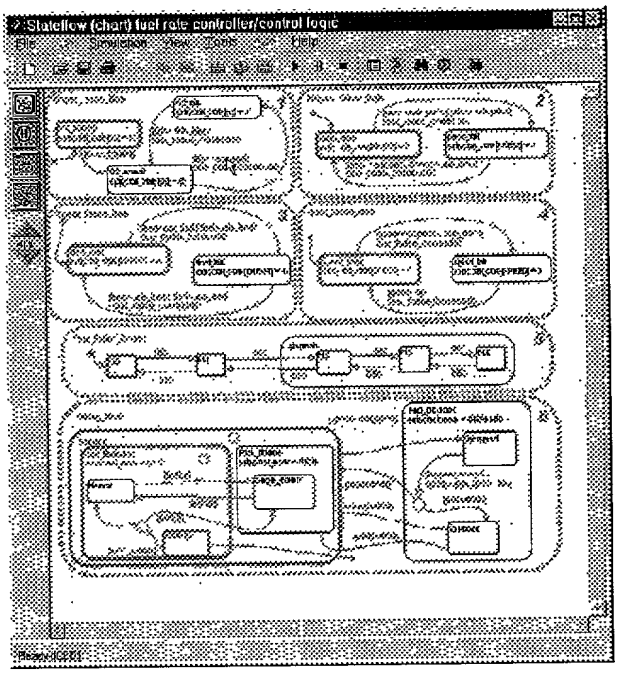
the throt_fail state. Transitions determine how states can change and can be guarded by conditions. For example, the throt_norm state can change to the throt_fail state when the measurement from the throttle sensor exceeds max_throt or is below min_throt.

The remaining two parallel states at the bottom consider the status of the four sensors simultaneously and determine the overall system operating mode. The Sens_Failure_Counter superstate acts as a store for the resultant number of sensor failures. This state is polled by the Fueling_Mode state that determines the fueling mode of the engine. If a single sensor fails, operation continues but the air/fuel mixture is richer to allow smoother running at the cost of higher emissions. If more than one sensor has failed, the engine shuts down as a safety measure, since the air/fuel ratio cannot be controlled reliably.

Although it is possible to run Stateflow charts asynchronously by injecting events from Simulink when required, the fueling control logic is polled synchronously at a rate of 100 Hz. Consequently, the sensors are checked every 1/100 second to see if they have changed status and the fueling mode adjusted accordingly.

Running the Model

On starting the simulation, and assuming no sensors have failed, the Stateflow diagram initializes in the Warmup mode in which the oxygen sensor is deemed to be in a warmup phase. If Stateflow is placed into animation mode, the current state of the system can clearly be seen highlighted in red on the Stateflow diagram, shown below.



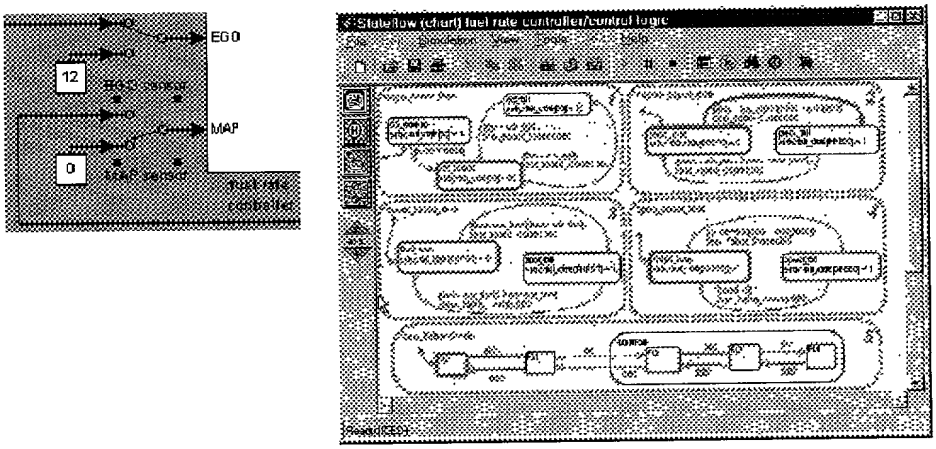
After a given time period, defined by o2_t_thresh, the sensor is deemed to have reached operating temperature and the system settles into the normal mode of operation, shown above, in which the fueling mode is set to NORMAL.

As the simulation progresses, the chart is woken synchronously every 0.01 second. The events and conditions that guard the transitions are evaluated and if a transition is valid, it is taken. The transition itself can be seen animated on the Stateflow diagram.

To illustrate this, we can provoke a transition by switching one of the sensors to a failure value on the top level Simulink model. The system detects throttle and pressure sensor failures when their measured values fall outside their

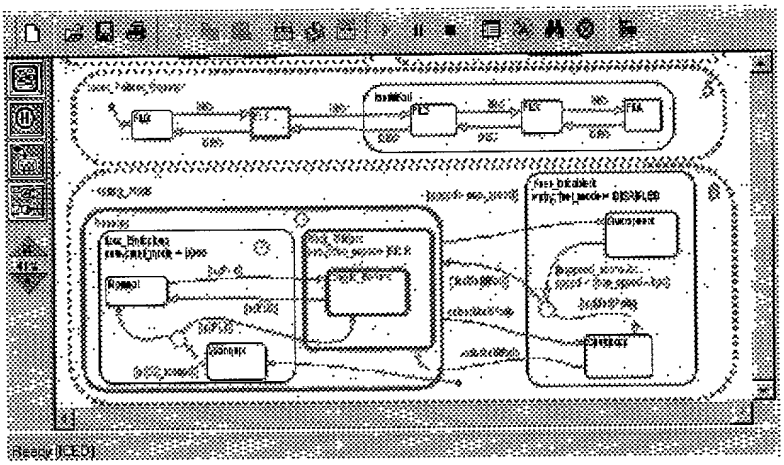
nominal ranges. A manifold vacuum in the absence of a speed signal is deemed to indicate a speed sensor failure. The oxygen sensor also has a nominal range for failure conditions but, because zero is both the minimum signal level and the bottom of the range, failure can be detected only when it exceeds the upper limit.

Switching the Simulink switch for the manifold air pressure (MAP) sensor causes a value of zero to be read by the fuel rate controller. When the chart is next woken up, the transition from the `press_norm` state becomes valid as the reading is now out of bounds and the transition is taken to the `press_fail` state. Regardless of which sensor fails, the model always generates the directed event broadcast `Sens_Failure_Counter.INC`. (thus making the triggering of the universal sensor failure logic independent of the sensor). This event causes a second transition from `FL0` to `FL1` to be taken in the `Sens_Failure_Counter` superstate. Note that both transitions can be seen animated on the Stateflow diagram below.

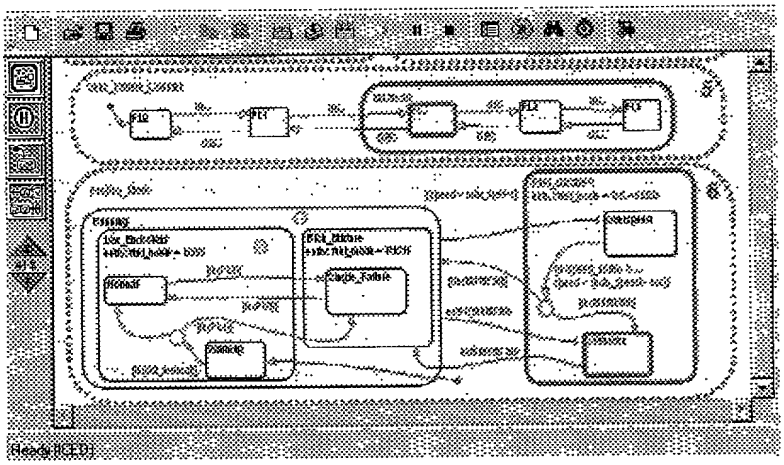


With the `Sens_Failure_Counter` state showing one failure, the condition that guards the transition from the `Low_Emissions.Normal` state to the `Rich_Mixture.Single_Failure` state is now valid and is therefore taken. As

the Fuel_Disabled state is entered, the, output fuel_mode is set to RICH, as shown below.



A second sensor failure causes the Sens_Failure_Counter to enter the Multifail state, broadcasting an implicit event which immediately triggers the transition from the Running state to the Shutdown state. On entering the Fuel_Disabled superstate the fueling_mode is DISABLED.

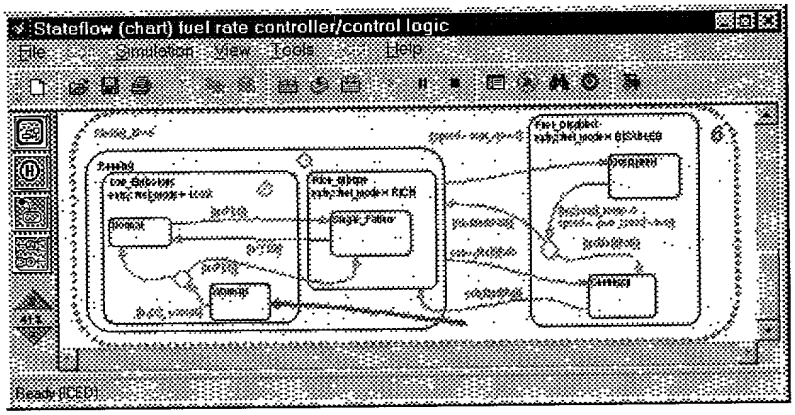


Implicit Event Broadcasts

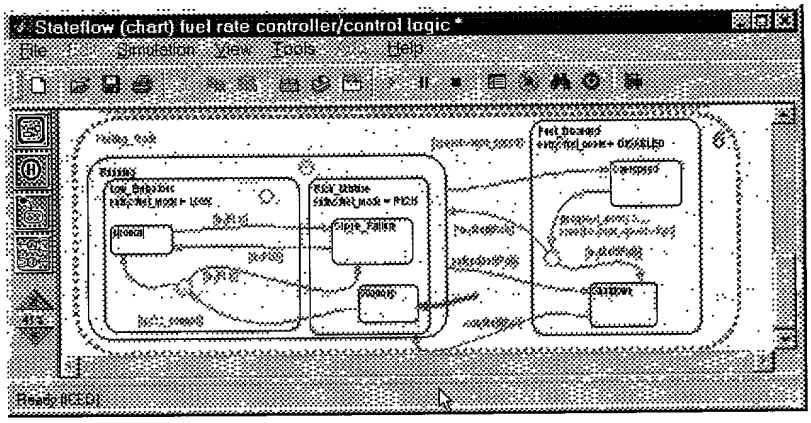
The fueling example above shows how the control logic can be represented in a clear and intuitive manner. The Stateflow diagram (or chart) has been developed in a way that allows the user, or a reviewer, to easily understand how the logic is structured. Implicit event broadcasts (such as `enter(multifail)`) and implicit conditions (`in(FLO)`) make the diagram easy to read and the generated code more efficient.

Modifying the Code

To illustrate how easy it is to modify the algorithm, consider the `Warmup` fueling state in the fuel control logic. At the moment the fueling is set to the low emissions mode.

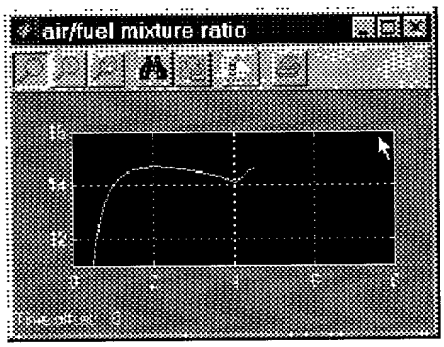


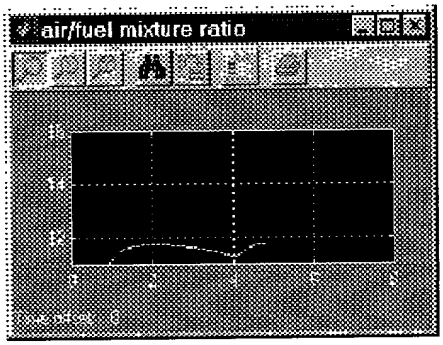
It may be decided that when the oxygen sensor is warming up, changing the warmup fueling mode to a rich mixture would be beneficial. In the Stateflow chart this can easily be achieved by changing the parent of the `Warmup` state to the `Rich_Mixture` state.



Once made, the alteration is obvious to all who need to inspect or maintain the code.

The results of changing the algorithm can be seen in the graphs of air/fuel mixture ratio for the first few seconds of engine operation after startup.





The left graph shows the air fuel ratio for the unaltered system whereas the right graph for the altered system shows how the air/fuel ratio stays low in the warming up phase indicating a rich mixture.

3

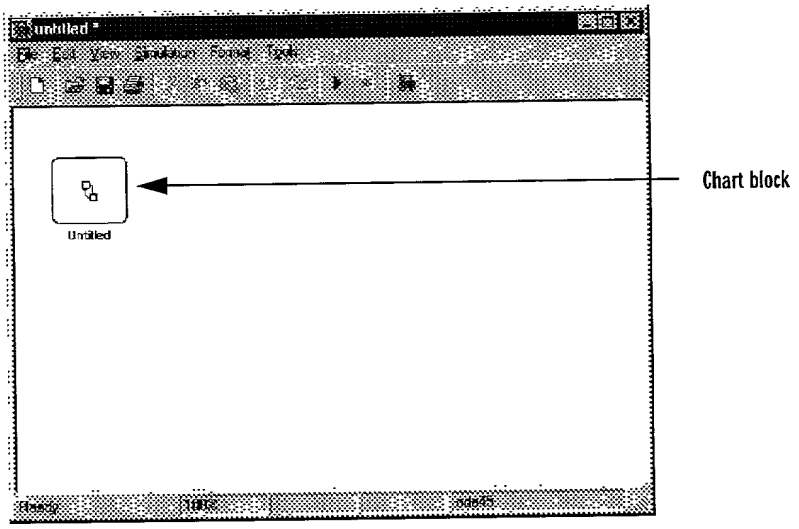
Creating Charts

Creating a Chart

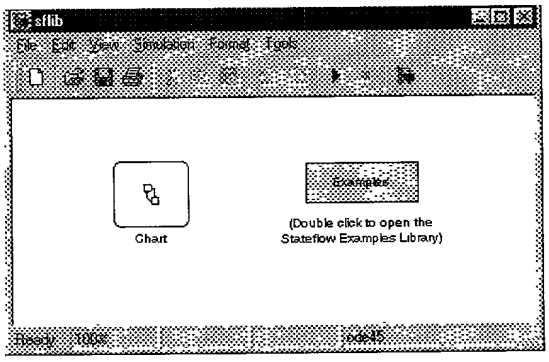
To create a Stateflow chart:

1 Create a new model with an empty chart block or copy an empty chart from the Stateflow block library into your model.

To create a new model with an empty chart, enter `sfnew` or `stateflow` at the MATLAB command prompt. The first command creates a new model.



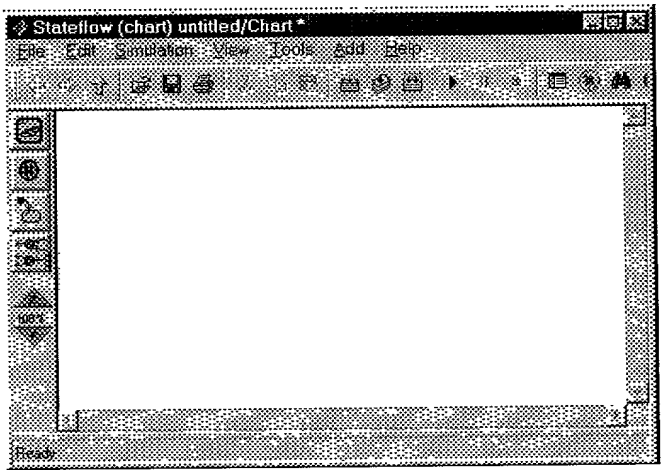
The second command also displays the Stateflow block library in case you want to create multiple charts in your model.



For information on creating your own chart libraries, see "Creating Chart Libraries" on page 3-54.

2 Open the chart by double-clicking on the chart block.

Stateflow opens the empty chart in a Stateflow editor window.



3 Use the Stateflow editor to draw and connect states representing the desired state machine or a component of the desired state machine.

See "Using the Stateflow Editor" on page 3-5 for more information.

4 Specify a wake up method for the chart.

See "Specifying Chart Properties" on page 3-30.

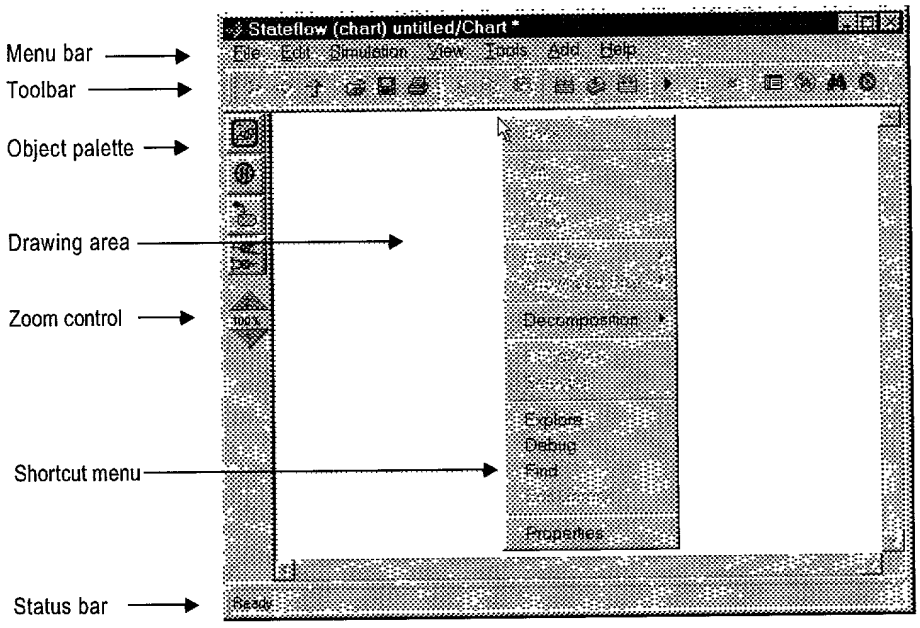
5 Interface the chart to other charts and blocks in your Stateflow model, using events and data.

See Chapter 4, "Defining Events and Data" and Chapter 5, "Defining Stateflow Interfaces" for more information.

6 Rename and save the model chart by selecting **Save Model As** from the Stateflow editor menu or **Save As** from the Simulink menu.

Using the Stateflow Editor

The Stateflow Editor consists of a window for displaying a state diagram and a set of commands that allow you to draw, zoom, modify, print, and save a state diagram displayed in the window.



The editor window includes the following elements:

- Menu bar

  Most editor commands are available from the menu bar.

- Toolbar

  Contains buttons for cut, copy, paste, and other commonly used editor commands. The toolbar also contains buttons for navigating a chart's subchart hierarchy (see "Navigating Subcharts" on page 3-46).

- Shortcut menus

  These menus pop up from the drawing area when you press the right mouse button. These menus display commands that apply only to the currently

selected object or to the chart as a whole, if no object is selected. See "Displaying Shortcut Menus" on page 3-6 for more information.

- Object Palette

  Displays a set of tools for drawing states, transitions, and other state chart objects. See "Drawing Objects" on page 3-6 for more information.

- Drawing area

  Displays an editable copy of a state diagram.

- Titlebar

  Displays the name of the state diagram being edited followed by an asterisk if the diagram needs to be saved.

- Zoom control

  See "Exploring Objects in the Editor Window" on page 3-12 for information on using the zoom control.

- Status bar

  Displays tooltips and status information.

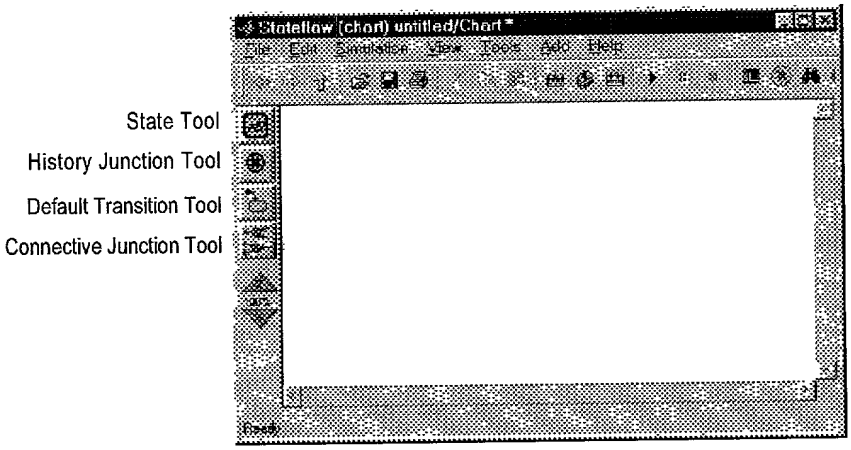
Displaying Shortcut Menus

Every object in a state diagram has a shortcut menu. To display the shortcut menu, move the cursor over the object and press the right mouse button. Stateflow then pops up a menu of operations that apply to the object. You can similarly display a shortcut menu for the chart as a whole. To display the chart shortcut menu, move the cursor to an unoccupied location in the diagram and press the right mouse button.

Drawing Objects

A state diagram comprises seven types of objects: states, boxes, functions, transitions, default transitions, history junctions, and connective junctions. Stateflow provides tools for creating instances of each of these types of objects. The Transition tool, used to draw transitions, is available by default. You select

and deselect the other tools by clicking their icons in the Stateflow editor's object palette.



You use the tools by clicking and dragging the cursor in the editor's drawing area. For more information, see the following topics:

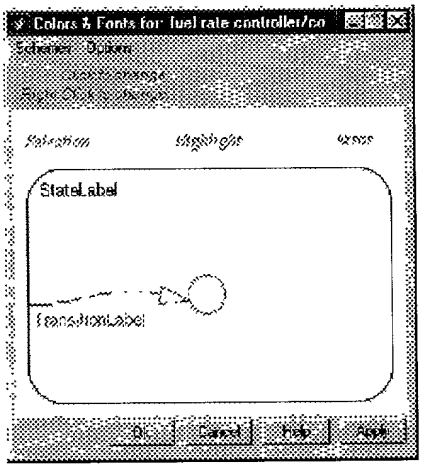
- "Creating States" on page 3-14
- "Creating Boxes" on page 3-21
- "Creating a Graphical Function" on page 3-34
- "Creating Transitions" on page 3-22
- "Creating Junctions" on page 3-27

Specifying Object Styles

An object's style consists of its color and the size of its label font. The Stateflow **Colors & Fonts** dialog allows you to specify a color scheme for a chart as a whole or colors and label fonts for various types of objects in a chart. To display the dialog, select **Style...** from the Stateflow editor's **Edit** menu. Stateflow displays the **Colors & Fonts** dialog. To specify the label font size of a particular object, select the object and choose the size from the **Set Font Size** submenu of the editor's **Edit** menu.

Colors & Fonts Dialog

The **Colors & Fonts Dialog** allows you to specify colors and label fonts for items in a chart or for the chart as a whole.

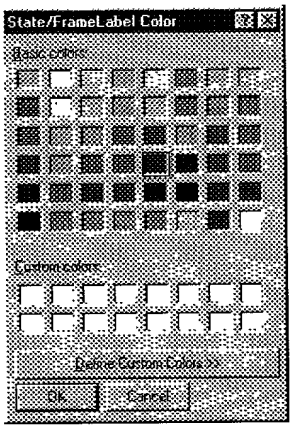


The drawing area of the dialog displays examples of the types of objects whose colors and font labels you can specify. The examples use the colors and label fonts specified by the current color scheme for the chart. To choose another color scheme, select the scheme from the dialog's **Schemes** menu. The dialog displays the selected color scheme. Choose **Apply** to apply the selected scheme to the chart or **Ok** to apply the scheme and dismiss the dialog.

To make the selected scheme the default scheme for all Stateflow charts, select **Make this the 'Default' scheme** from the dialog's **Options** menu.

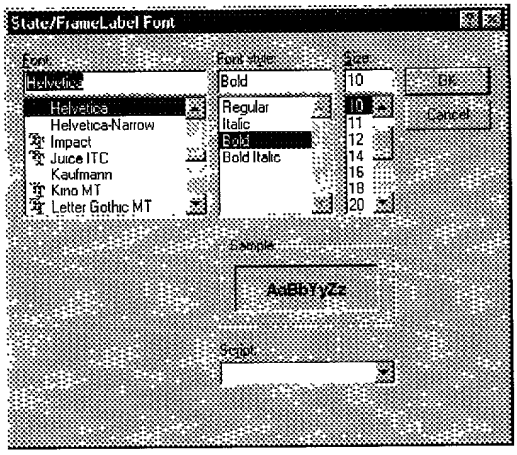
To modify the current scheme, position the cursor over the example of the type of object whose color or label font you want to change. Then click the left mouse button to change the object's color or the right mouse button to change the

object's font. If you click the left mouse button, Stateflow displays a color chooser dialog.



Use the dialog to select a new color for the selected object type.

If the selected object is a label and you click the right mouse button, Stateflow displays a font selection dialog.



Use the font selector to choose a new font for the selected label.

To save changes to the default color scheme, select **Save defaults to disk** from the **Colors & Fonts** dialog's **Options** menu.

Note Choosing **Save defaults to disk** has no effect if the modified scheme is not the default scheme.

Selecting and Deselecting Objects

Once an object is in the drawing area, you need to select it to make any changes or additions to that object. To select an object, click on it. When an object is selected, it is highlighted in the color set as the selection color (blue by default; see "Specifying Object Styles" on page 3-7 for more information).

To select multiple objects, click the left mouse button and drag the selection rubberband so that the rubberband box encompasses or touches the objects you want to select. Once all objects are within the rubberband, release the left mouse button. All objects or portions of objects within the rubberband are selected.

Simultaneously pressing the **Shift** key and clicking on an object either adds that object to the selection list if it was deselected or deselects the object if it is on the selection list. This is useful to select objects within a state without selecting the state itself.

To select all objects in the Stateflow diagram, choose **Select All** from the **Edit** menu or the right mouse button shortcut menu.

Simultaneously, pressing the **Shift** key and doing a rubberband selection selects objects touched by the rubberband if they are deselected and deselects objects touched by the rubberband if they are selected.

Pressing the **Escape** key deselects all selected objects. Pressing the **Escape** key again displays the parent of the current chart.

Cutting and Pasting Objects

You can cut one or more objects from the drawing area or cut and then paste the object(s) as many times as you like. You can cut and paste objects from one Stateflow diagram to another. The Stateflow clipboard contains the most recently cut selection list of objects. The object(s) are pasted in the drawing area location closest to the current mouse location.

To cut an object, select the object and choose **Cut** from either:

- The **Edit** menu on the main window
- The right mouse button shortcut menu

Pressing the **Ctrl** and **X** keys simultaneously is the keyboard equivalent to the **Cut** menu item.

To paste the most recently cut selection list of objects, choose **Paste** from either:

- The **Edit** menu on the main window
- The right mouse button shortcut menu

Pressing the **Ctrl** and **V** keys simultaneously is the keyboard equivalent to the **Paste** menu item.

Copying Objects

To copy and paste an object in the drawing area, select the object(s), click and hold the right mouse button down, and drag to the desired location in the drawing area. This operation also updates the Stateflow clipboard.

Alternatively, to copy from one Stateflow diagram to another, choose the **Copy** and then **Paste** menu items from either:

- The **Edit** menu on the Stateflow graphics editor window
- Any right mouse button shortcut menu

Pressing the **Ctrl** and **C** keys simultaneously is the keyboard equivalent to the **Copy** menu item. States that contain other states (superstates) can be grouped together.

Editing Object Labels

Some Stateflow objects (e.g., states and transitions) have labels. To change these labels, place the cursor anywhere in the label and click. The cursor changes to an I-beam. You can then edit the text.

Changing a Label's Font Size

The shortcut menus allows you to change a label's font size:

1 Select the state(s) whose label font size you want to change.

2 Click the mouse's right button to display the shortcut menu.

3 Place the cursor over the **Font Size** menu item.

A menu of font sizes appears.

4 Select the desired font size from the menu.

Stateflow changes the font size of all labels on all selected states to the selected size.

Exploring Objects in the Editor Window

To view or modify events and data defined by any state visible in the Stateflow editor window (see Chapter 4, "Defining Events and Data"), position the editor cursor over the state, display the state's context menu (by pressing the right mouse button), and select **Explore** from the context menu. Stateflow opens the Stateflow Explorer (if not already open) and expands its object hierarchy view (see "Explorer Main Window" on page 6-3) to show any events or data defined by the state.

To view events and data defined by a transition or junction's parent state, select **Explore** from the transition or junction's context menu.

Zooming a Diagram

You can magnify or shrink a diagram, using the following zoom controls:

- **Zoom Factor Selector**. Selects a zoom factor (see "Using the Zoom Factor Selector").
- **Zoom In** button. Zooms in by the current zoom factor.
- **Zoom Out** button. Zooms out by the current zoom factor.

Using the Zoom Factor Selector

The **Zoom Factor Selector** allows you to specify the zoom factor by:

- Choosing a value from a menu.

  Click on the selector to display the menu.

- Double-clicking on the **Zoom Factor Selector** selects the zoom factor that will fit the view to all selected objects or all objects if none are selected.

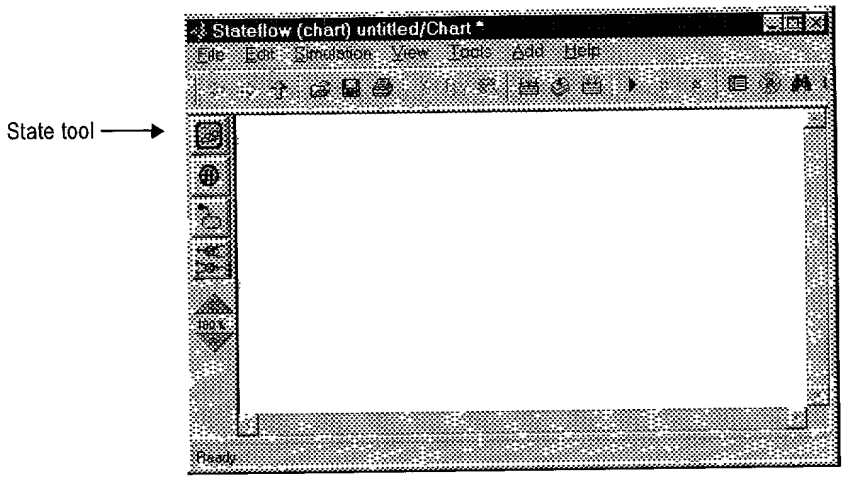
  You can achieve the same effect by choosing **Fit to View** from any shortcut menu or by pressing the **F** key to apply the maximum zoom that includes all selected objects. Press the space bar to fit all objects to the view.

- Clicking on the **Zoom Factor Selector** and dragging up or down.

  Dragging the mouse upward increases the zoom factor. Dragging the mouse downward decreases the zoom factor. Alternatively, right-clicking and dragging on the percentage value resizes while you are dragging.

Creating States

You create states by drawing them in the Stateflow editor's drawing area, using the Stateflow's State tool.



To activate the State tool, click or double-click on the **State** button in the Stateflow toolbar. Single-clicking on the button puts the State tool in single-creation mode. In this mode, you create a state by clicking the tool in the drawing area. Stateflow creates the state at the specified location and returns to edit mode.

Double-clicking on the **State** button puts the State tool in multiple-creation mode. This mode works the same way as single-creation mode except that the State tool remains active after you create a state. You can thus create as many states as you like in this mode without having to reactivate the State tool. To return to edit mode, click on the **State** button, or right click in the drawing area, or press the **Escape** key.

To delete a state, select it and choose **Cut** (**Ctrl+X**) from the **Edit** or any shortcut menu or press the **Delete** key.

Moving and Resizing States

To move a state, select, drag, and release it in a new position. To resize a state, drag one of the state's corners. When the cursor is over a corner, it appears as

a double-ended arrow (PC only; cursor appearance will vary on other platforms).

Creating Substates

A substate is a state that can be active only when another state, called its parent, is active. States that have substates are known as superstates. To create a substate, click the State tool and drag a new state into the state you want to be the superstate. Stateflow creates the substate in the specified parent state. You can nest states in this way to any depth. To change a substate's parentage, drag it from its current parent in the state diagram and drop it in its new parent.

Note A parent state must be large enough to accommodate all its substates. You may therefore need to resize a parent state before dragging a new substate into it.

Grouping States

Grouping a state causes Stateflow to treat the state and its contents as a graphical unit. This simplifies editing a state diagram. For example, moving a grouped state moves all its substates as well. To group a state, double-click on it or its border or select **Grouped** from the **Make Contents** submenu on the state or box shortcut menu. Stateflow thickens the state's border and grays its contents to indicate that it is grouped. To ungroup a state, double-click it or its border or select **Ungrouped** from the **Make Contents** submenu units shortcut menu. You need to ungroup a superstate to select objects within the superstate.

Specifying State Decomposition

Stateflow allows you to specify whether activating a state activates all or only one of its substates. A state whose substates are all active when it is active is said to have parallel (AND) decomposition. A state in which only one substate is active when it is active is said to have exclusive (OR) decomposition. An empty state's decomposition is exclusive. You can alter a state's decomposition only if it contains substates. To alter a state's decomposition, select the state, press the right mouse button to display the state's shortcut menu, and choose either **Parallel (AND)** or **Exclusive (OR)** from the menu.

You can also specify the state decomposition of a chart. In this case, Stateflow treats the chart's top-level states as substates of the chart. Stateflow creates states with exclusive decomposition. To specify a chart's decomposition, deselect any selected objects, press the right mouse button to display the chart's shortcut menu, and choose either **Parallel (AND)** or **Exclusive (OR)** from the menu.

The appearance of a superstate's substates indicates the superstate's decomposition. Exclusive substates have solid borders, parallel substates, dashed borders. A parallel substate also contains a number in its upper right corner. The number indicates the activation order of the substate relative to its sibling substates.

Specifying Activation Order for Parallel States

You specify the activation order of parallel states by arranging them from top-to-bottom and left-to-right in the state diagram. Stateflow activates the states in the order in which you arrange them. In particular, a top-level parallel state activates before all the states whose top edges reside at a lower level in the state diagram. A top-level parallel state also activates before any other state that resides to the right of it at the same vertical level in the diagram. The same top-to-bottom, left-to-right activation order applies to parallel substates of a state.

Note Stateflow displays the activation order of a parallel state in its upper right corner.

Labeling States

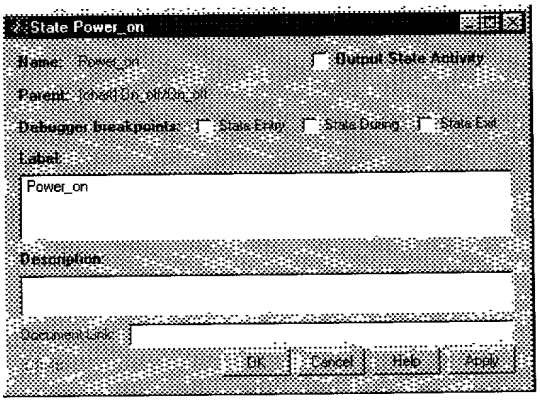
Every state has a label. A state's label specifies its name and actions that a state machine takes when entering or exiting the state or while the state is active. Initially, a state's label is empty. Stateflow indicates this by displaying a ? in the state's label position (upper left corner). Click on the label or display the state's properties dialog (see "Using the State Properties Dialog Box" on page 3-17) to add to or change its contents. For more information on labeling states, see the following topics:

- "Naming States" on page 3-18

- "Defining State Actions" on page 3-18

Using the State Properties Dialog Box

The **State Properties** dialog box lets you view and change a state's properties. To display the dialog for a particular state, choose **Properties** from the state's shortcut menu or click on the state's entry in the Explorer content pane. Stateflow displays the **State Properties** dialog box.



The dialog includes the following fields.

Field	Description
Name	Stateflow diagram name; read-only; click on this hypertext link to bring the state to the foreground.
Output State Activity	Check this box to cause Stateflow to output the activity status of this state to Simulink via a data output port on the chart block containing the state. See "Outputting State Activity to Simulink" on page 3-20 for more information.
Parent	Parent of this state; a / character indicates the Stateflow diagram is the parent; read-only; click on this hypertext link to bring the parent to the foreground.

Field	Description
Debugger breakpoints	Click on the check boxes to set debugging breakpoints on state `entry`, `during`, or `exit` actions. See Chapter 10, "Debugging" for more information.
Label	The state's label. See the section titled "Labeling States" on page 3-16 for more information.
Description	Textual description/comment.
Document Link	Enter a URL address or a general MATLAB command. Examples are: `www.mathworks.com`, `mailto:email_address`, `edit /spec/data/speed.txt`.

Click on the dialog **Apply** button to save the changes. Click on the **Revert** button to cancel any changes and return to the previous settings. Click on the **Close** button to save the changes and close the dialog box. Click on the **Help** button to display the Stateflow online help in an HTML browser window.

Naming States

Naming a state allows a state machine to reference the state. To name a state, enter the state's name in the first line of the state's label. Names are case-sensitive. To avoid naming conflicts, do not assign the same name to sibling states. However, you can assign the same name to states that do not share the same parent.

Defining State Actions

Stateflow allows you to specify actions that occur when a state machine enters a state, exits a state, and while a state is active.

Defining Entry Actions

An entry action is an action executed by a state machine when it enters a particular state as the result of taking a transition to that state. To specify the entry action to be taken for a given state, add an entry block to the state's label. An `entry` block begins on a new line and consists of the entry action keyword, `entry` or `en`, followed by a colon, followed by one or more action statements on

one or more lines. You must separate statements on the same line by a comma or semicolon. See "Action Language" on page 7-37 for information on writing action statements.

Note You can also begin a state's entry action on the same line as the state's name. In this case, begin the entry action with a forward slash (/) instead of the entry keyword.

Defining Exit Actions

An exit action is an action executed by a state machine when it exits a state as the result of taking a transition away from the state or the occurrence of an event (see "Defining On-Event Actions" below). To specify an exit action for a state, add an exit block to the state's label. The format of an exit block is the same as that of an entry block except that the exit block begins with the keyword `exit` or `ex`.

Defining During Actions

A during action is an action that a state machine executes while a state is active, that is, after the state machine has entered the state and while there is no valid transition away from the state. To specify a during action, add a during block to the state's label. A during block has the same format as an entry block except that it begins with the keyword `during` or `dur`.

Defining On-Event Actions

An on-event action is an action that a state machine takes when a state is active and one or more events of a specific type occur. (See "Defining Events" on page 4-2 for information on defining and using events to drive a state machine.) To specify an event handler for a state, add an on-event block to the state. An on-event block has the same format as an entry action block except that it begins with the keyword, `on`, followed by the name of the event, followed by a colon, for example

```
on ev1: exit();
```

A state machine can respond to multiple events, with either the same or different actions, when a state is active. If you want more than one type of event to trigger the same action, specify the keyword as `on events`, where

events is a comma-separated list of the events that trigger the actions, for example,

```
on ev1, ev2: exit();
```

If you want different events to trigger different actions, enter multiple event blocks in the state's label, each specifying the action for a particular event or set of events, for example,

```
on ev1: action1(); on ev2: action2();
on ev3, ev4: exit();
```

Note Use a during block to specify actions that you want a state machine to take in response to any visible event that occurs while the machine is in a particular state (see "Defining During Actions" on page 3-19).

Outputting State Activity to Simulink

Stateflow allows a chart to output the activity of its states to Simulink via a data port on the state's chart block. To enable output of a particular state's activity, first name the state (see "Naming States" on page 3-18), if unnamed, then check the **Output State Activity** check box on the state's property dialog (see "Using the State Properties Dialog Box" on page 3-17). Stateflow creates a data output port on the chart block containing the state. The port has the same name as the state. Stateflow also adds a corresponding data object of type State to the Stateflow data dictionary. During simulation of a model, the port outputs 1 at each time step in which the state is active; 0, otherwise. Attaching a scope to the port allows you to monitor a state's activity visually during the simulation. See "Defining Output Data" on page 4-21 for more information.

Note If a chart has multiple states with the same name, only one of those states can output activity data. If you check the `Output State Activity` property for more than one state with the same name, Stateflow outputs data only from the first state whose `Output State Activity` property you specified.

Creating Boxes

Stateflow allows you to use graphical entities called boxes to organize your diagram visually. To create a box, first create a superstate containing the objects to be boxed. Then, select **Box** from the superstate's **Type** shortcut menu. Stateflow converts the superstate to a box, redrawing its border with sharp corners to indicate its changed status.

Boxes are primarily graphical entities. They do not correspond to any real element of a state machine. However, boxes do affect the activation order of a diagram's parallel states. In particular, a box wakes up before any parallel states or boxes that are lower down or to the right of the box in the diagram. Within a box, parallel states still wake up in top down, right-to-left order.

You can group and ungroup boxes and hide or show them in the same way you hide or show states. See "Grouping States" on page 3-15 and "Working with Subcharts" on page 3-42 for more information.

Creating Transitions

Place the cursor at a straight portion of the border of the source state. Click the border when the cursor changes to a cross-hair. Drag and release on a straight portion of the border of the destination state when the transition changes from a dotted line to a solid line. The solid line indicates the transition is in position to be attached. The cross-hair will not appear if you place the cursor on a corner, since corners are used for resizing.

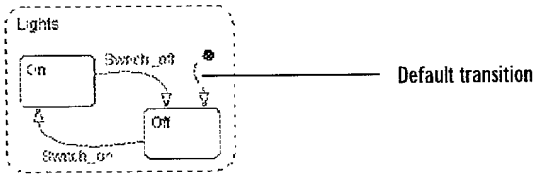
Use a similar procedure to create transitions between junctions. You can start or end a transition at any point on a junction. To draw a perfectly straight transition between two junctions, hold the **Shift** key down as you draw the transition. If you draw a nearly straight transition between two junctions without holding down the **Shift** key, Stateflow straightens the transition after you finish drawing the transition.

To delete a transition, select it and choose **Cut** (**Ctrl+X**) from the **Edit** menu or any shortcut menu or press the **Delete** key.

What Is a Default Transition?

The default transition object is a transition with a destination, but no source object. Default transitions specify which exclusive (OR) state is to be active when there is ambiguity between two or more exclusive (OR) states at the same level in the hierarchy. Default transitions also specify that a junction should be entered by default.

In the `Lights` subsystem, the default transition to the `Lights.Off` substate indicates that when the `Lights` superstate becomes active, the `Off` substate becomes active by default.



Default transitions specify which exclusive (OR) substate in a superstate the system enters by default, in the absence of any information. History junctions

override default transition paths in superstates with exclusive (OR) decomposition.

Creating Default Transitions

Click on the **Default transition** button in the toolbar  , and click on a location in the drawing area close to the state or junction you want to be the destination for the default transition. Drag the mouse to the destination object to attach the default transition. The size of end point of the default transition is proportional to the arrowhead size. Default transitions can be labeled just like other transitions. See the section titled "Labeling Default Transitions" on page 7-21 for an example.

Editing Transition Attach Points

Place the cursor over an attach point until it changes to a small circle. Click and drag the mouse to move the attach point; release to drop the attach point. You can edit both the source and destination attach points of a transition.

The appearance of the transition changes from a solid to a dashed line when editing a destination attach point. Once you attach the transition to a destination, the dashed line changes to a solid line. The appearance of the transition changes to a default transition when editing a source attach point.

To delete a transition, select it and choose **Cut (Ctrl+X)** from the **Edit** or any shortcut menu, or press the **Delete** key.

Labeling Transitions

The ? character is the default empty label for transitions. Transitions and default transitions follow the same labeling format. Select the transition to display the ? character. Click on the ? to edit the label.

Transition labels have this general format.

```
event [condition]{condition_action}/transition_action
```

Specify, as appropriate, relevant names for event, condition, condition_action, and transition_action. Transitions do not have to have labels. You can specify some, all, or none of the parts of the label.

Label Field	Description
event	Event that causes the transition to be evaluated.
condition	Defines what, if anything, has to be true for the condition action and transition to take place.
condition_action	If the condition is true, the action specified executes and completes.
transition_action	After a valid destination is found and the transition is taken, this action executes and completes.

To apply and exit the edit, deselect the object.

See these sections in Chapter 7, "Notations" for more information:

- "Transitions" on page 7-14
- "Action Language" on page 7-37

Valid Labels

Labels can consist of any alphanumeric and special character combination, with the exception of embedded spaces. Labels cannot begin with a numeric character. The length of a label is not restricted. Carriage returns are allowed in most cases. Within a condition, you must specify an ellipsis (...) to continue on the next line.

Changing Arrowhead Size

The arrowhead size is a property of the destination object. Changing one of the incoming arrowheads of an object causes all incoming arrowheads to that object to be adjusted to the same size. The arrowhead size of any selected transitions, and any other transitions ending at the same object, is adjusted.

To adjust arrowhead size from the **Transition** shortcut menu:

1 Select the transition(s) whose arrowhead size you want to change.

2 Place the cursor over a selected transition, click the right mouse button to display the shortcut menu.

A menu of arrowhead sizes appears.

3 Select an arrowhead size from the menu.

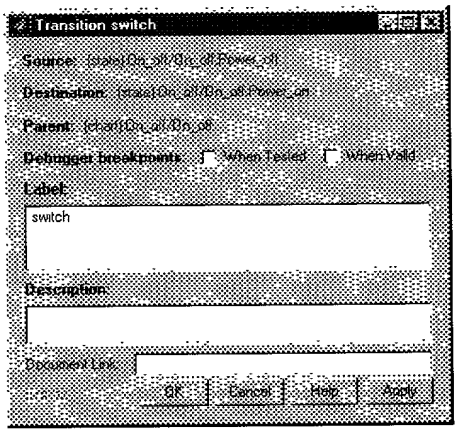
Moving Transition Labels

You can move transition labels to make the Stateflow diagram more readable. To move a transition label, click on and drag the label to the new location and then release the mouse button.

If you mistakenly click and release the left mouse button on the label, you will be in edit mode. Press the **Esc** key to deselect the label and try again.

Using the Transition Properties Dialog

Select a transition and click the right mouse button on that transition's border to display the **Transition** shortcut menu. Choose **Properties** to display the **Transition properties** dialog box.



This table lists and describes the transition object fields.

Field	Description
Source	Source of the transition; read-only; click on the hypertext link to bring the transition source to the foreground.
Destination	Destination of the transition; read-only; click on the hypertext link to bring the transition destination to the foreground.
Parent	Parent of this state; read-only; click on the hypertext link to bring the parent to the foreground.
Debugger breakpoints	Click on the check boxes to set debugging breakpoints either when the transition is tested for validity or when it is valid.
Label	The transition's label. See the section titled "Labeling a Transition" on page 7-15 for more information on valid label formats.
Description	Textual description/comment.
Document Link	Enter a Web URL address or a general MATLAB command. Examples are: `www.mathworks.com`, `mailto:email_address`, `edit/spec/data/speed.txt`.

Click on the **Apply** button to save the changes. Click on the **OK** button to save the changes and close the dialog box. Click on the **Cancel** button to close the dialog without applying any changes made since the last time you clicked the **Apply** button. Click on the **Help** button to display the Stateflow online help in an HTML browser window.

Creating Junctions

To create one junction at a time, click on a **Junction** button in the toolbar and click on the desired location for the junction in the drawing area. To create multiple junctions, double-click on the **Junction** button in the toolbar. The button is now in multiple object mode. Click anywhere in the drawing area to place a junction in the drawing area. Additional clicks in the drawing area create additional junctions. Click on the **Junction** button or press the **Esc** key to cancel the operation.

You can choose from these types of junctions.

Name	Button Icon	Description
Connective junction		One use of a connective junction is to handle situations where transitions out of one state into two or more states are taken based on the same event but guarded by different conditions.
History junction		Use a history junction to indicate, when entering this level in the hierarchy, that the last state that was active becomes the next state to be active.

Changing Size

To adjust the junction size from the **Junction** shortcut menu:

1 Select the junction(s) whose size you want to change. The size of any selected junctions is adjusted.

2 Place the cursor over a selected junction, click the right mouse button to display the shortcut menu and place the cursor over **Junction Size**.

A menu of junction sizes appears.

3 Select a junction size from the menu.

Changing Arrowhead Size

The arrowhead size is a property of the destination object. Changing one of the incoming arrowheads of a junction causes all incoming arrowheads to that junction to be adjusted to the same size. The arrowhead size of any selected junctions is adjusted.

To adjust arrowhead size from the **Junction** shortcut menu:

1 Select the junction(s) whose incoming arrowhead size you want to change.

2 Place the cursor over a selected junction, click the right mouse button to display the shortcut menu. Place the cursor over **Arrowhead Size.**

A menu of arrowhead sizes appears

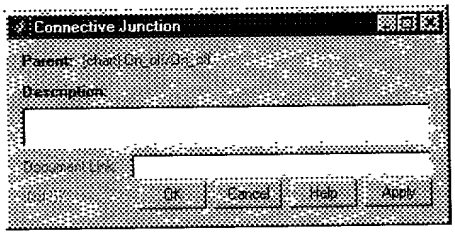
3 Select a size from the menu.

Moving a Junction

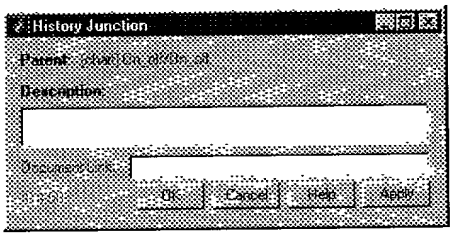
To move a junction, select, drag, and release it in a new position.

Editing Junction Properties

Select a junction, click the right mouse button on that junction to display the **Junction** shortcut menu. Choose **Properties** to display the **Connective Junction Properties** dialog box.



This is the **History Junction Properties** dialog box.



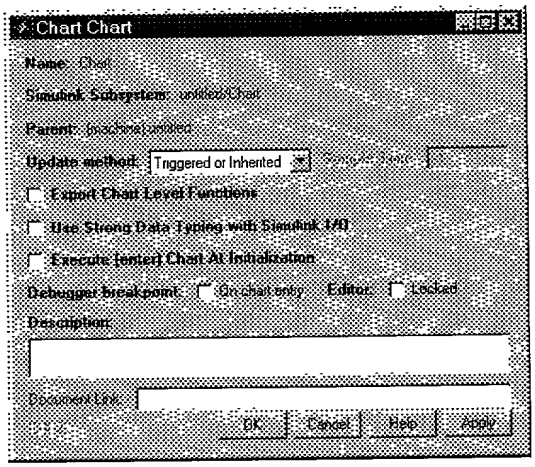
This table describes the junction object fields.

Field	Description
Parent	Parent of this state; read-only; click on the hypertext link to bring the parent to the foreground.
Description	Textual description/comment.
Document Link	Enter a URL address or a general MATLAB command. Examples are: `www.mathworks.com`, `mailto:email_address`, `edit/spec/data/speed.txt`.

Click on the **Apply** button to save the changes. Click on the **Cancel** button to cancel any changes since the last apply. Click on the **OK** button to save the changes and close the dialog box. Click on the **Help** button to display the Stateflow online help in an HTML browser window.

Specifying Chart Properties

Click the right mouse button in an open area of the Stateflow diagram to display the **General** shortcut menu. This is the **Chart properties** dialog box.



This table lists and describes the chart object fields.

Field	Description
Name	Stateflow diagram name; read-only; click on this hypertext link to bring the chart to the foreground.
Simulink Subsystem	Simulink subsystem name; read-only; click on this hypertext link to bring the Simulink subsystem to the foreground.
Parent	Parent name; read-only; click on this hypertext link to display the parent's property dialog box.
Update method	Choose from `Triggered` or `Inherited`, `Sampled`, or `Continuous`.

Field	Description
Export Chart Level Functions	Exports graphical functions defined at the chart's root level. See "Exporting Graphical Functions" on page 3-39 for more information.
Use Strong Data Typing with Simulink IO	If this option is checked, this chart block can accept and output signals of any data type supported by Simulink. The type of an input signal must match the type of the corresponding chart input data item (see "Defining Input Data" on page 4-20). Otherwise, a type mismatch error occurs. If this item is unchecked, this chart accepts and outputs only signals of type `double`. In this case, Stateflow converts Simulink input signals to the data types of the corresponding chart input data items. Similarly, Stateflow converts chart output data (see "Defining Output Data" on page 4-21) to `double`, if this option is unchecked.
Execute (enter) Chart at Initialization	Check this option if you want a chart's state configuration to be initialized at time 0 instead of at the first occurrence of an input event.
Sample Time	If **Update method** is **Sampled**, enter a sample time.
Debugger breakpoint	Click on the check box to set a debugging breakpoint **On chart entry.**
Editor	Click on the **Locked** check box to mark the Stateflow diagram as read-only and prohibit any write operations.
Description	Textual description/comment.
Document Link	Enter a Web URL address or a general MATLAB command. Examples are: `www.mathworks.com`, `mailto:email_address`, `edit/spec/data/speed.txt`.

Click on the **Apply** button to save the changes. Click on the **Cancel** button to cancel any changes since the last apply. Click on the **OK** button to save the changes and close the dialog box. Click on the **Help** button to display the Stateflow online help in an HTML browser window.

Waking Up Charts

Stateflow lets you specify the method by which a simulation updates (wakes up) a chart. To specify a wake up method for a chart, set the chart's `Update method` property (see "Specifying Chart Properties" on page 3-30) to one of the following options:

- `Triggered or Inherited`

  This is the default update method. Specifying this method causes inputs from the Simulink model to determine when the chart wakes up during a simulation. If you define input events for the chart (see "Defining Input Events" on page 4-7), the chart awakens when trigger signals appear on the chart's trigger port. If you define data inputs (see "Defining Input Data" on page 4-20) but no event inputs, the chart awakens at the rate of the fastest data input. If you do not define any inputs for the chart, the chart wakes up at the model's solver sample rate.

- `Sampled`

  Simulink awakens (samples) the Stateflow block at the rate you specify as the block's `Sample Time` property. An implicit event is generated by Simulink at regular time intervals corresponding to the specified rate. The sample time is in the same units as the Simulink simulation time. Note that other blocks in the Simulink model may have different sample times.

- `Continuous`

  The Stateflow block wakes up at each step in the simulation, as well as at intermediate time points that may be requested by the Simulink solver.

See "Defining the Interface to External Sources" on page 5-23 and *Using Simulink* for more information.

Working with Graphical Functions

A *graphical function* is a function defined by a flow graph. Graphical functions are similar to textual functions, such as MATLAB and C functions. Like textual functions, graphical functions can accept arguments and return results. You invoke graphical functions in transition and state actions in the same way you invoke MATLAB and C functions. Unlike C and MATLAB functions, however, graphical functions are full-fledged Stateflow objects. You use the Stateflow editor to create them and they reside in your Stateflow model along with the diagrams that invoke them. This makes graphical functions easier to create, access, and manage than textual functions, whose creation requires external tools and whose definitions reside separately from the model.

Creating a Graphical Function

To create a graphical function:

1 Create a state in your model where you want the function to appear.

A function can reside anywhere in a diagram, either at the top level or within any state or subchart. The location of a function determines its scope, that is, the set of states and transitions that can invoke the function. In particular, the scope of a function is the scope of its parent state or chart, with the following exceptions.

- The chart containing the function exports its graphical functions.

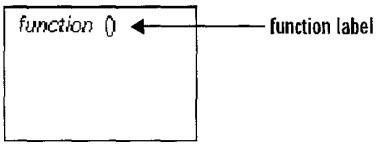
  In this case, the scope of the function is the scope of its parent state machine. See "Exporting Graphical Functions" on page 3–39 for more information.

- A child of the function's parent define a function of the same name.

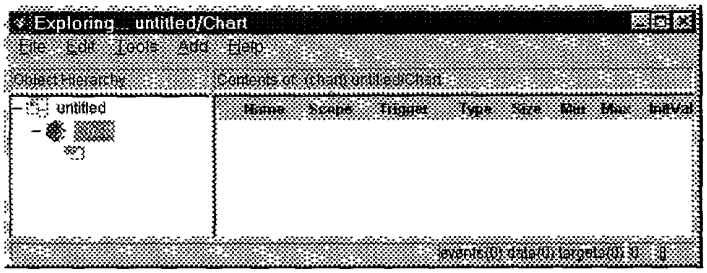
  In this case, the function defined in the parent is not visible anywhere in the child or its children. In other words, a function defined in a state or subchart shadows any functions of the same defined in the ancestors of that state or subchart.

2 Select **Function** from the **Type** submenu of the newly created state's shortcut menu.

Stateflow converts the state to a graphical function.



The new function appears as an unnamed object in the Stateflow Explorer.

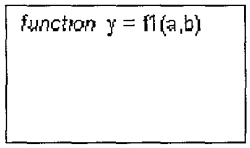


3 Enter a function prototype in the function label.

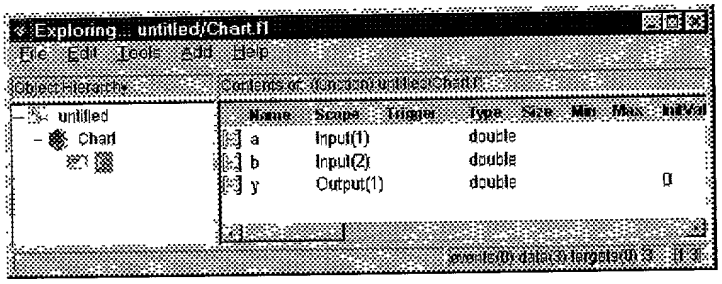
The function prototype specifies a name for the function and formal names for its arguments and return value. A prototype has the syntax

$y = f(a_1, a_2, \ldots a_n)$

where f is the function's name, a_1, a_2, a_n are formal names for its arguments, and y is the formal name for its return value. The following example shows a prototype for a graphical function named f1 that takes two arguments and returns a value.



The return values and arguments that you declare in the prototype appear in the Explorer as data items parented by the function object.



The **Scope** field in the Explorer indicates the role of the corresponding argument or return value. Arguments have scope `Input`. Return values have scope `Output`. The number that appears in parentheses for the scope of each argument is the order in which the argument appears in the function's prototype. When a Stateflow action invokes a function, it passes arguments to the function in the same order.

In the context of graphical function prototypes, the term *scope* refers to the role (argument or return value) of the data items specified by the function's prototype. The term scope can also refer to a data item's visibility. In this sense, arguments and return values have local scope. They are visible only in the flow diagram that implements the function.

Note You can use the Stateflow editor to change the prototype of a graphical function at any time. When you are done editing the prototype, Stateflow updates the data dictionary and the Explorer to reflect the changes.

---

4 Specify the data properties (data type, initial value, etc.) of the function's arguments and return values (if it has any).

See"Setting Data Properties" on page 4–14 for information on setting data properties. The following restrictions apply to argument and return value properties.

- A function cannot return more than one value.
- Arguments and return values cannot be arrays.

- Arguments cannot have initial values.
- Arguments must have scope `Input`.
- Return values must have scope `Output`.

5 Create any additional data items that the function may need to process when it is invoked.

See "Adding Data to the Data Dictionary" on page 4–13 for information on how to create data items. A function can access only items that it owns. Thus, any items that you create for use by the function must be created as children of the function. The items that you create can have any of the following scopes.

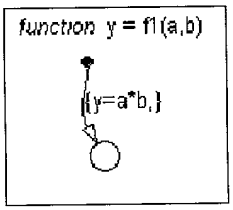
- `Local`
  A local data item persists from invocation to invocation. For example, if the item is equal to 1 when the function returns from one invocation, the item will equal 1 the next time the function is invoked.
- `Temporary`
  Stateflow creates and initialize a copy of a temporary item for each invocation of the function.
- `Constant`
  A constant data items retains its initial value through all invocations of the function.

Note You can also assign `Input` and `Output` scope to data items that you create (i.e, to items that do not correspond to the function's formal arguments and return value). However, Input and Output items that do not correspond to your function's formal arguments and return values will cause parse errors. In other words, you cannot create arguments or return values by creating data items.

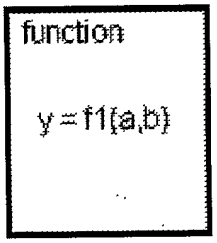
All data items (other than arguments and return values) parented by a graphical function can be initialized from the workspace. However, only local items can be saved to the workspace.

6 Create a flow diagram within the function that performs the action to be performed when the function is invoked.

At a minimum, the flow diagram must include a default transition terminated by a junction. The following example shows a minimal flow diagram for a graphical function that computes the product of its arguments.



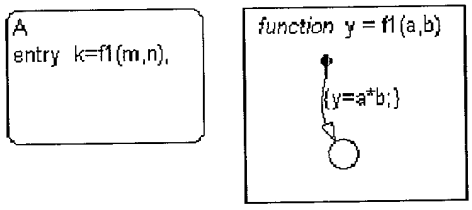
7 If you prefer, hide the function's contents by selecting **Subcharted** from the **Make Contents** submenu of the function's shortcut menu.



Invoking Graphical Functions

Any state or transition action that is in the scope of a graphical function can invoke that function. The invocation syntax is the same as that of the function prototype, with actual arguments replacing the formal parameters specified in the prototype. If the data types of the actual and formal argument differ, Stateflow casts the actual argument to the type of the formal parameter. The

following example shows a state entry action that invokes a function that returns the product of its arguments.



Exporting Graphical Functions

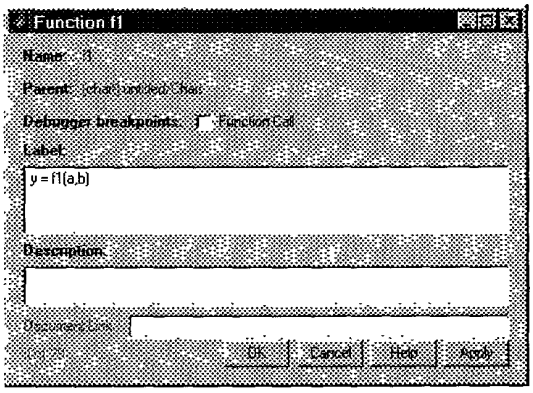
You can export a chart's root-level graphical functions. Exporting the functions extends their scope to include all other charts in the same model. To export a chart's root-level functions, check **Export Chart Level Functions** on the chart's **Chart Properties** dialog box (see "Specifying Chart Properties" on page 3-30).

When parsing a chart, Stateflow does not check to see whether the chart's usage of exported functions is correct. It is thus up to you to see ensure that the chart passes arguments of the correct type to an exported function and assigns the return value of the function to a variable of the correct type. Failure to use the function correctly can cause link or runtime errors.

Note You cannot export functions from a chart library.

Specifying Graphical Function Properties

A graphical function has properties that you can specify. To specify the properties, choose properties from the function's shortcut menu. The Function properties dialog box appears.



The dialog has the following fields.

Field	Description
Name	Function name; read-only; click on this hypertext link to bring the function to the foreground.
Parent	Parent of this function; a / character indicates the Stateflow diagram is the parent; read-only; click on this hypertext link to bring the parent to the foreground.
Debugger breakpoints	Click on the check box to set a breakpoint where the function is called. See Chapter 10, "Debugging" for more information.
Label	The function's label. Specifies the function's prototype. See "Creating a Graphical Function" on page 3-34 for more information.

Field	Description
Description	Textual description/comment.
Document Link	Enter a URL address or a general MATLAB command. Examples are: `www.mathworks.com`, `mailto:email_address`, `edit/spec/data/speed.txt`.

Working with Subcharts

Stateflow allows you to create charts within charts. A chart that is embedded in another chart is called a *subchart*. The subchart can contain anything a top-level chart can, including other subcharts. In fact, you can nest subcharts to any level.

A subchart appears as a labeled block in the chart that contains it. A subchart is itself a superstate of the states and charts that it contains. You can define actions and default transitions for subcharts just as you can for superstates. You can also create transitions to and from subcharts just as you can create transitions to and from superstates. Further, you can create transitions from states residing outside a subchart to any state within a subchart, and vice versa. The term *super transition* refers to a transition that crosses subchart boundaries in this way (see "Working with Supertransitions" on page 3-48 for more information).

Subcharts enable you to reduce a complex chart to a set of simpler, hierarchically organized diagrams. This makes the chart easier to understand and maintain. Nor do you have to worry about changing the semantics of the chart in any way. Stateflow ignores subchart boundaries when simulating and generating code from Stateflow models.

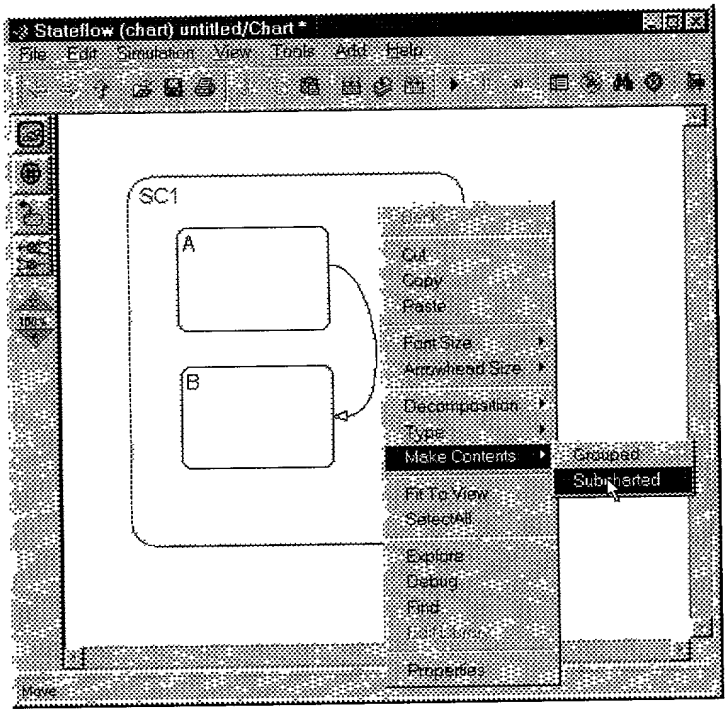
Subcharts define a containment hierarchy within a top-level chart. A subchart or top-level chart is said to be the *parent* of the charts it immediately contains. A subchart or a top-level chart is said to be an *ancestor* of all the subcharts contained by its children and their descendents.

Creating a Subchart

You create a subchart by converting an existing state, box, or graphical function into the subchart. The object to be converted can be one that you have created expressly for the purpose of making a subchart or it can be an existing object whose content you want to turn into a subchart.

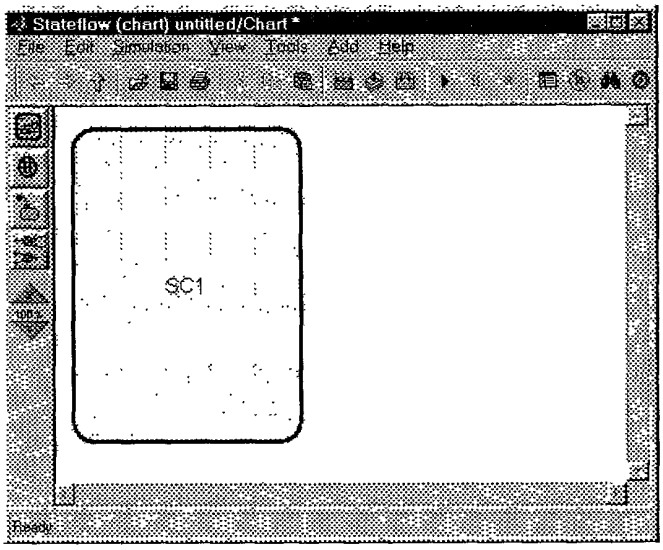
To convert a new or existing state, box, or graphical function to a subchart:

1 Select the object and click your mouse's right button to display the Stateflow shortcut menu.



2 Select **Subcharted** from the **Make Contents** menu.

Stateflow converts the selected state, graphical function, or box to a subchart.



Note When you convert a box to a subchart, the subchart retains the attributes of a box. In particular, the resulting subchart's position in the chart determines its activation order (see "Creating Boxes" on page 3-21 for more information).

To convert the subchart back to its original form, select the subchart and uncheck the **Subcharted** item of the **Make Contents** submenu of the Stateflow shortcut menu.

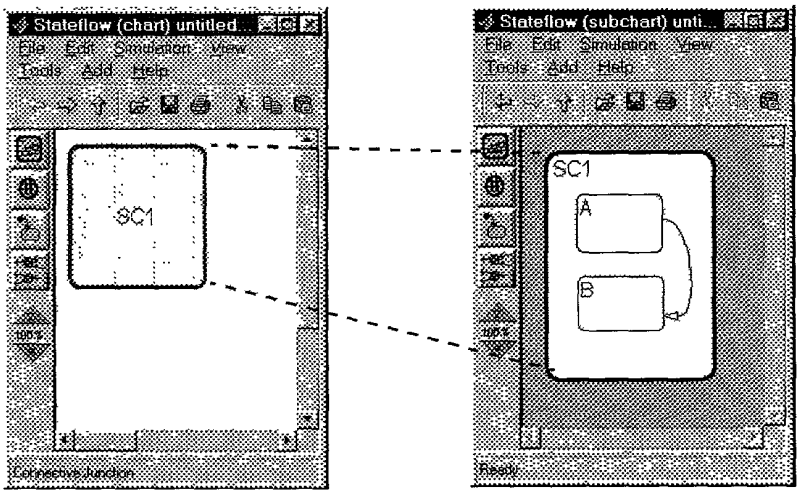
Manipulating Subcharts as Objects

Subcharts are first-class objects in Stateflow. You can use the same techniques to drag, copy, cut, paste, relabel, and resize subcharts as you use to perform similar objects on states and boxes. You can also draw transitions to and from

a subchart and any other state or subchart at the same or different levels in the chart hierarchy (see "Working with Supertransitions" on page 3-48).

Opening a Subchart

Opening a subchart allows you to view and change its contents. To open a subchart, double-click your mouse anywhere in the block that represents the subchart. Stateflow replaces the current contents of the editor window with the contents of the subchart.

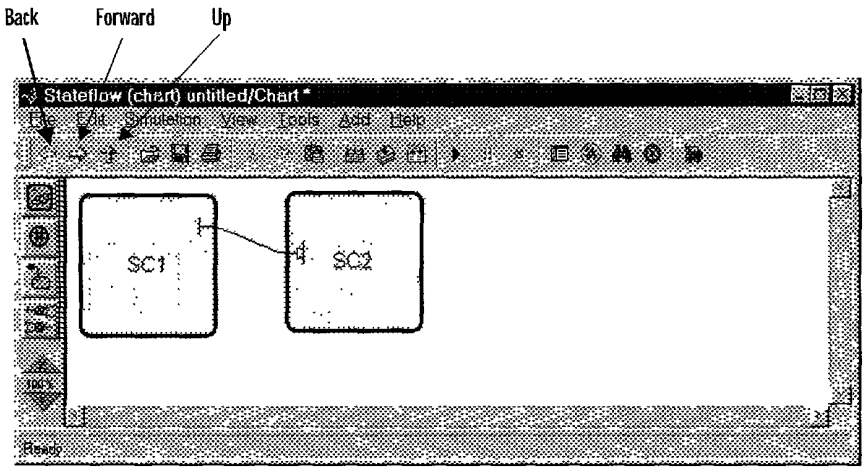


A shaded border surrounds the contents of the subchart. Stateflow uses the border to display supertransitions.

To return to the previous view, select **Back** from the Stateflow shortcut menu, press the **Esc** key on your keyboard, or select the up or back arrow on the Stateflow toolbar.

Navigating Subcharts

The Stateflow toolbar contains a set of buttons for navigating a chart's subchart hierarchy.



- Up

  If the Stateflow editor is displaying a subchart, this button replaces the subchart with the subchart's parent. If the editor is displaying a top-level chart, this button raises the Simulink model window containing the chart.

  The next two buttons allow you to retrace your steps as you navigate up and down a subchart hierarchy.

- Back

  Returns to the chart that you visited before the current chart.

- Forward

  Returns to the chart that you visited after visiting the current chart.

Editing a Subchart

You can perform any editing operation on a subchart that you can perform on a top-level chart. You can create, copy, paste, cut, relabel, resize, and group states, transitions, and other subcharts. You can also create transitions among states and junctions in a subchart in the same way you create them among

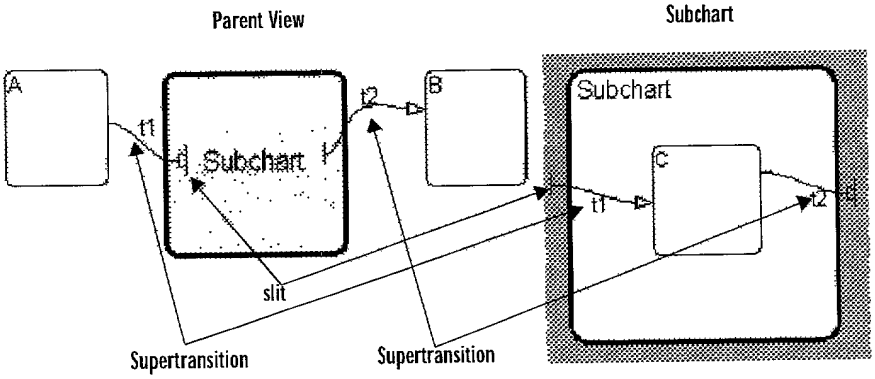
states in a top-level chart. (See "Working with Supertransitions" on page 3-48 for information on creating transitions to and from a subchart). It is also possible to cut-and-paste objects between different levels in your chart. For example, to copy objects from a top-level chart to one of its subcharts, first open the top-level chart and copy the objects. Then open the subchart and paste the objects into the subchart.

Working with Supertransitions

About Supertransitions

A *supertransition* is a transition between different levels in a chart, for example, between a state or junction in a top-level chart and a state or junction in one of its subcharts or between states residing in different subcharts at the same or different level in a diagram. Stateflow allows you to create supertransitions that span any number of levels in your chart, for example, from a junction at the top-level to a state that resides in a subchart several layers deep in the chart.

The point where a supertransition enters or exits a subchart is called a *slit*. Slits divide a supertransition into graphical segments. For example, the following diagram shows two super transitions as seen from the perspective of a subchart and its parent chart, respectively.



In this example, supertransition t1 goes from state A in the parent chart to state C in the subchart and supertransition t2 goes from state C in the subchart to state B in the parent chart. Note that both segments of t1 and t2 have the same label.

Drawing a Supertransition

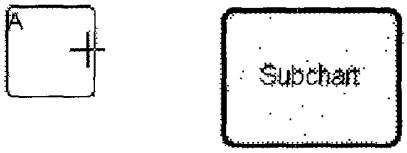
The procedure for drawing a supertransition differs slightly, depending on whether you are drawing the transition from an object outside a subchart to an object inside the chart, or vice versa.

Drawing a Transition Into a Subchart

To draw a supertransition from an object outside a subchart to an object inside the subchart:

1 Position the mouse cursor over the border of the state.

The cursor assumes a crosshair shape.

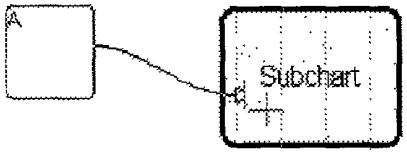


2 Drag the mouse.

Dragging the mouse causes a supertransition segment to appear. The segment looks like a regular transition. It is curved and is tipped by an arrowhead.

3 Drag the segment's tip anywhere just inside the border of the subchart.

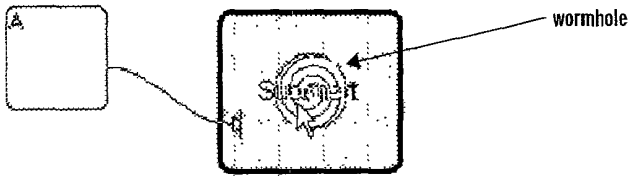
The arrowhead now penetrates the slit.



If you are not happy with the initial position of the slit, you can continue to drag the slit around the inside edge of the subchart to the desired location.

4 Continue dragging the cursor toward the center of the subchart.

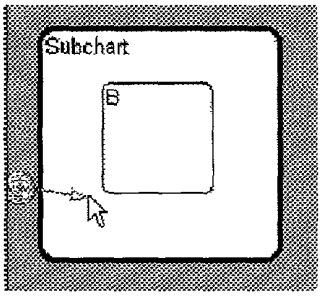
A wormhole appears in the center of the subchart.



A *wormhole* allows you to open a subchart while drawing a supertransition.

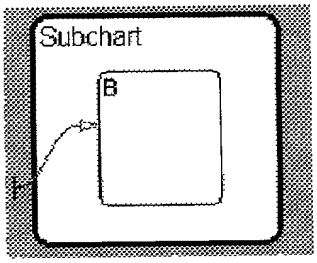
5 Drag the mouse pointer over the center of the wormhole.

The subchart opens. Now the wormhole and supertransition are visible inside the subchart.



6 Drag and drop the tip of the supertransition anywhere on the border of the object that you want to terminate the transition.

This completes the drawing of the supertransition.



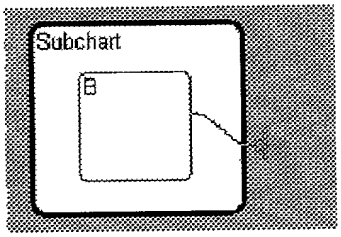
Note If the terminating object resides within a subchart in the current subchart, simply drag the tip of the supertransition through the wormhole of the inner subchart and complete the connection inside the inner chart. You can draw a supertransition to an object at any depth in the chart in this fashion.

Drawing a Transition Out of a Subchart

To draw a supertransition out of a subchart:

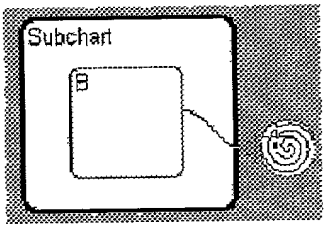
1 Draw an inner transition segment from the source object anywhere just outside the border of the subchart

A slit appears.



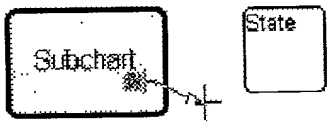
2 Keep dragging the transition away from the border of the subchart.

A wormhole appears.



3 Drag the transition down the wormhole.

The parent of the subchart appears.



4 Complete the connection.

Note If the parent chart is itself a subchart and the terminating object resides at a higher level in the subchart hierarchy, you can continue drawing by dragging the supertransition into the border of the parent subchart. This allows you to continue drawing the supertransition at the higher level. In this way, you can connect objects separated by any number of layers in the subchart hierarchy.

Labeling Supertransitions

To label a supertransition, label any of its segments using the same procedure used to label a regular transition (see "Labeling Transitions" on page 3-23). The resulting label appears on all segments of the transition. If you change the label on any segment, the change appears on all segments.

Creating Chart Libraries

A Stateflow chart library is a Simulink block library that contains Stateflow chart blocks (and, optionally, other types of Simulink blocks as well). Just as Simulink libraries serve as repositories of commonly used blocks, chart libraries serve as repositories of commonly used charts.

You create a chart library in the same way you create other types of Simulink libraries. First, create an empty chart library by selecting **Library** from the **New** submenu of Simulink's **File** menu. Then create or copy chart blocks into the library just as you would create or copy chart blocks into a Stateflow model.

You use chart libraries in the same way you use other types of Simulink libraries. To include a chart from a library in your Stateflow model, copy or drag the chart from the library to the model. Simulink creates a link from the instance in your model to the instance in the library. This allows you to update all instances of the chart simply by updating the library instance.

Note Events parented by a library state machine are invalid. Stateflow allows you to define such events but flags them as errors when parsing a model.

Stateflow Printing Options

The following options are available for printing Stateflow models:

- You can print a block diagram of the Stateflow model, using the Simulink **Print** command.

  The Simulink print command is labeled **Print...** on the Stateflow editor's **File** menu. See the *Using Simulink* manual or online Simulink documentation for more information on the command.

- You can print the current view of a diagram, using the Stateflow **Print Current View** command.

  See "Printing the Current View" on page 3-55.

- You can generate a report that documents the Stateflow component of a Stateflow model, using the Stateflow **Print Book** command.

  See "Printing a Stateflow Book" on page 3-56.

- You can generate a report that documents an entire Stateflow model, including both Simulink and Stateflow components, using the Simulink Report Generator.

  The Simulink Report Generator is available as a separate product. See the *Report Generator User's Guide* for more information.

Printing the Current View

To print a Stateflow diagram, open the chart containing the diagram and select **Print Current View** from the Stateflow editor's **File** menu. Stateflow displays a submenu of printing options.

- **To File**

  Converts the current view to a graphics file. Selecting this option displays a submenu of graphics file formats. Choose the desired format to convert the current view to a file in that format.

- **To Clipboard**

  Copies the current view to the system clipboard. Selecting this option displays a submenu of graphics formats. Select a format to copy the current view to the clipboard in that format.

- **To Figure**

  Converts the current view to a MATLAB figure window.

- **To Printer**

  Prints the current view on the current printer.

You can also print the current view, using the `sfprint` command. See `sfprint` in Chapter 11, "Function Reference" for more information about printing from the command line.

Printing a Stateflow Book

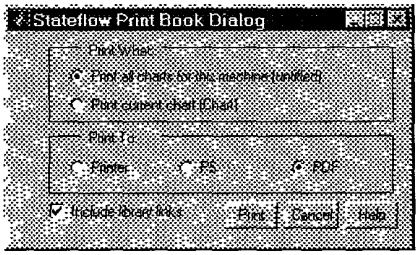
A Stateflow book is a report that documents all the elements of a Stateflow chart, including states, transitions, junctions, events, and data. You can generate a book documenting a specific chart or all charts in a model.

To generate a Stateflow book:

1 Select and open one of the charts you want to document.

2 Select **Print Book** from the Stateflow editor's **File** menu.

Stateflow displays the **Print Book** dialog box.



3 Check the desired print options on the dialog.

4 Select the **Print** button to generate the report.

Configuring a Target

Configuring a target entails some or all of the following steps:

1 Add the target, if necessary, to the state machine's target list.

  See "Adding a Target to a State Machine's Target List" on page 9-9 for instructions on how to add targets to a state machine's target list.

2 Specify code generation options.

  See "Specifying Code Generation Options" on page 9-11 for more information.

3 Specify custom code options.

  See "Specifying Custom Code Options" on page 9-17 for more information.

4 Check "Apply to all Libraries" on the **Target Builder** dialog box if you want the selected options to apply to the code generated for charts imported from chart libraries.

Configuring an RTW target may require additional steps. See the *Real-Time Workshop User's Guide* for more information.

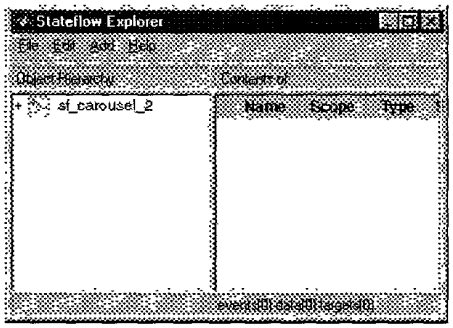
Adding a Target to a State Machine's Target List

Building an Real-Time Workshop target requires that you first add the target to the list of potential targets maintained by Stateflow for a particular model.

To add a target:

1 Select **Explore** from the Stateflow editor's **Tools** menu.

The Stateflow Explorer appears.



The Explorer object hierarchy shows the state machines currently loaded in memory.

2 Select the state machine to which you want to add the Real-Time Workshop target.

The Explorer displays the selected state machine's data, events, and targets in the contents pane.

3 Select **Target** from the Explorer's **Add** menu to add a target with the default name "untitled" to the selected machine.

4 Rename the target.

You must name the target `rtw`. (A state machine can have only one Real-Time Workshop target.)

Renaming the Target

To rename the target:

1 Select the target in the Explorer's content pane and press the right mouse button.

  A pop-up menu appears.

2 Select **Rename** from the pop-up menu.

  The Explorer redisplays the selected target's name in an edit box.

3 Change the target's name in the edit box.

4 Click outside the edit box to close it.

Specifying Code Generation Options

Specifying code generation options differs slightly depending on whether you are specifying options for a simulation target or an RTW target.

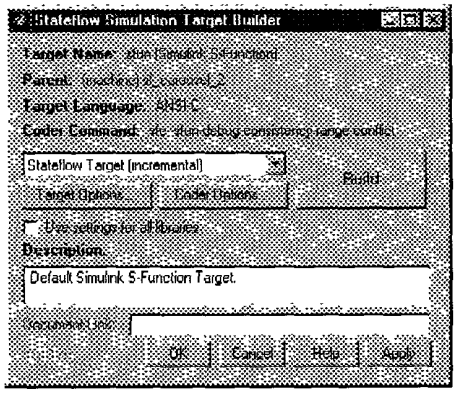
Simulation Target

To specify code generation options for a simulation target:

1 Open the target builder dialog for the target.

  You can do this by selecting **Open Simulation Target** from the graphics editor's **Tools** menu or by clicking on the target in the Stateflow Explorer.

The **Simulation Target Builder** dialog box for the simulation target appears.



2 Select **Coder Options...**.

The **Simulation Coder Options** dialog box appears (see "Simulation Coder Options Dialog Box" on page 9-14).

3 Check the desired options.

4 Select **Apply** to apply the selected options or **OK** to apply the options and close the dialog.

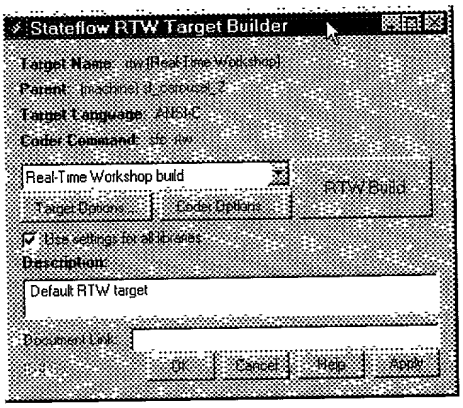
RTW Target

To specify code generation options for an RTW target:

1 Open the target builder dialog for the RTW target.

You can do this by selecting **Open RTW Target** from the graphics editor's **Tools** menu or by clicking on the target in the Stateflow Explorer.

The **RTW Target Builder** dialog box for the simulation target appears.



2 Select **Coder Options...**.

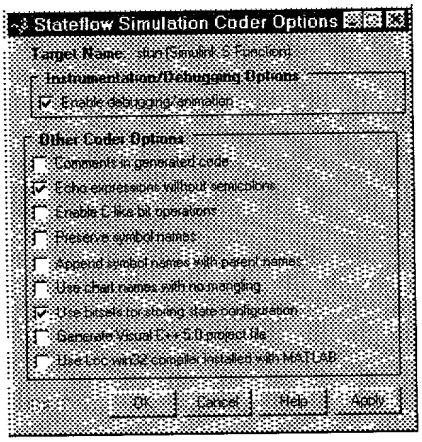
The **RTW Coder Options** dialog box appears (see "RTW Coder Options Dialog Box" on page 9-15).

3 Check the desired options.

4 Select **Apply** to apply the selected options or **OK** to apply the options and close the dialog.

Simulation Coder Options Dialog Box

The Stateflow simulation coder provides the following options.



Enable Debugging/Animation. Enables chart animation and debugging. Stateflow enables debugging code generation when you use the debugger to start a model simulation. You can enable or disable chart animation separately in the debugger. (The Stateflow debugger does not work with stand-alone and RTW targets. Therefore, Stateflow and Real-Time Workshop do not generate debugging/animation code for these targets, even if this option is enabled.)

Comments in generated code. Include comments from generated code.

Echo expressions without semicolons. Display runtime output in the MATLAB command window, specifically actions that are not terminated by a semicolon.

Enable C-like bit operations. Recognize C bit-wise operators (~, &, |, ^, >>, etc.) in action language statements and encode these operators as C bit-wise operations.

Preserve symbol names. Preserve symbol names (names of states and data) when generating code. This is useful when the target contains custom code that accesses state machine data. Note that this option can generate duplicate C symbols if the source chart contains duplicate symbols, for example, two substates with identical names. Enable the next option to avoid duplicate substate names.

Append symbol names with parent names. Generates a state or data name by appending the name of the item's parent to the item's name.

Use chart names with no mangling. Exports the names of generated functions so that they can be invoked by user-written C code.

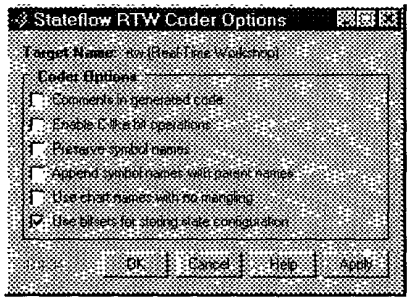
Use bitsets for storing state configuration. Use bitsets for storing state configuration variables. This can significantly reduce the amount of memory required to store the variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets.

Generate Visual C++ 5.0 project file. Generates a Microsoft Visual C++ 5.0 project file for the simulation target. This simplifies use of Visual C++ to debug targets that include custom code.

Use Lcc-win32 compiler installed with MATLAB. Use the lcc compiler to build this target. See "Setting Up Build Tools on Windows" on page 9-5 for more information. (This option appears only on the Windows version of Stateflow.)

RTW Coder Options Dialog Box

The **RTW Coder Options** dialog box provides the following options.



Comments in generated code. Include comments in the generated code.

Enable C-like bit operations. Recognize C bit-wise operators (~, &, |, ^, >>, etc.) in action language statements and encode these operators as C bit-wise operations.

Preserve symbol names. Preserve symbol names (names of states and data) when generating code. This is useful when the target contains custom code that accesses state machine data. Note that this option can generate duplicate C symbols if the source chart contains duplicate symbols, for example, two substates with identical names. Enable the next option to avoid duplicate substate names.

Append symbol names with parent names. Generates a state or data name by appending the name of the item's parent to the item's name.

Use chart names with no mangling. Exports the names of generated functions so that they can be invoked by user-written C code.

Use bitsets for storing state configuration. Use bitsets for storing state configuration variables. This can significantly reduce the amount of memory required to store the variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets.

Specifying Custom Code Options

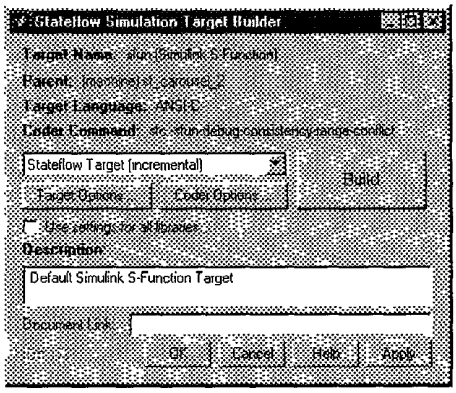
You must specify various configuration options (see "Custom Code Options" on page 9-18) to build custom code into a simulation target.

To specify the custom code options:

1 Open the **Target Builder** dialog box for the target in which you want to include custom code.

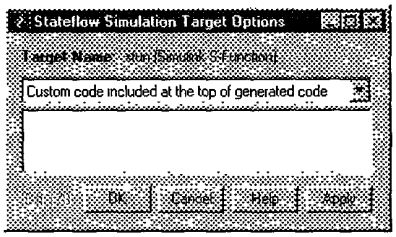
You can do this by selecting the appropriate open target item (e.g., **Open Simulation Target**) from the Stateflow editor's **Tools** menu or by clicking on the simulation target in the Stateflow Explorer.

The **Target Builder** dialog box appears, for example,



2 Select **Target Options** from the dialog.

The **Target Options** dialog box appears.



The dialog box contains a drop-down list listing various options for specifying what code to include in the target and where the code is located. The edit box below the list displays the setting for the current option.

3 Select the options required to specify your code and enter the specifications in the edit box.

See "Custom Code Options" on page 9-18 for information on how to use these options to specify your custom code.

4 Select **Apply** to apply the specification to the target or **OK** to apply the specifications and close the dialog.

Custom Code Options

The target options dialog provides the following options for specifying custom code to be built into a simulation target:

Custom code included at the top of generate code. Custom C code to be included at the top of a generated header file that is included at the top of all generated source code files. In other words, all generated code sees code specified by this option. Use this option to include header files that declare custom functions and data used by generated code.

Custom include directory paths. Space-separated list of paths of directories containing custom header files to be included either directly (see first option above) or indirectly in the compiled target.

Custom source files. Space separated list of source files to be compiled and linked into the target.

Note Stateflow ignores the preceding two options when building RTW targets. This means that all source files required for building custom code into an RTW target must reside in MATLAB's working directory.

Custom libraries. Space-separated list of libraries containing custom object code to be linked into the target.

Custom make files. Space-separated list of custom makefiles. The Stateflow code generator includes these makefiles at the head of the makefile it generates to build the simulation target. You can use this option to include makefiles for building custom code required by the target.

Build command. The MATLAB command used to build the target.

Code command. The MATLAB command used to invoke the code generator (`sfc`, by default). You can add command-line arguments for `sfc` options not reflected on the **Coder Options** dialog box for the target.

Custom initialization code. Code statements that are executed once at the start of simulation. You can use this initialization code to invoke functions that allocate memory or perform other initializations of your custom code.

Custom termination code. Code statements that are executed at the end of simulation. You can use this code to invoke functions that free memory allocated by custom code or perform other cleanup tasks.

Parsing

Parser

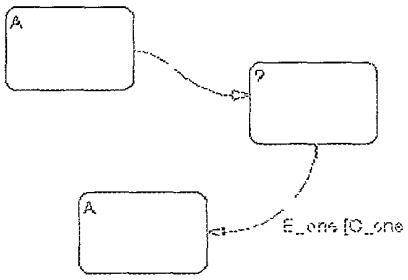
The parser evaluates the graphical and nongraphical objects in each Stateflow machine against the supported Stateflow notation and the action language syntax.

Parse the Machine or the Stateflow Diagram

Explicitly parse each Stateflow diagram in the machine by choosing **Parse** from the graphics editor **Tools** menu. Explicitly parse the current Stateflow diagram by choosing **Parse Diagram** from the graphics editor **Tools** menu. The machine is implicitly parsed when you simulate a model, build a target, or generate code.

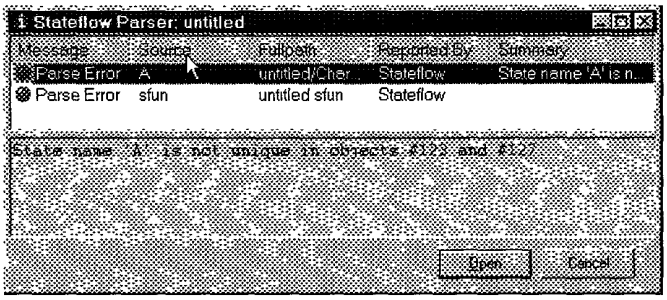
In all cases, a pop-up information window is displayed when the parsing is complete. If the parsing is unsuccessful, one error at a time is displayed (in red) in the informational window. The Stateflow diagram automatically selects and pans to the object containing the parse error. Double-click on the error in the information window to bring the Stateflow diagram to the forefront, zoom (fit to view), and select the object containing the parse error. Press the space bar to zoom back out. Fix the error and reparse the Stateflow diagram. Informational messages are also displayed in the MATLAB command window.

These steps describe parsing, assuming this Stateflow diagram.



1 Parse the Stateflow diagram.

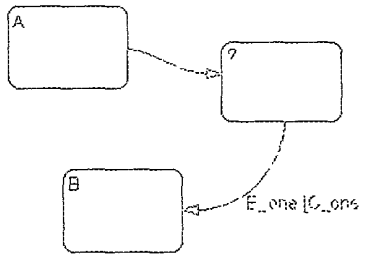
Choose **Parse Diagram** from the graphics editor **Tools** menu to parse the Stateflow diagram. State A in the upper left-hand corner is selected and this message is displayed in the pop-up window and the MATLAB command window.



2 Fix the parse error.

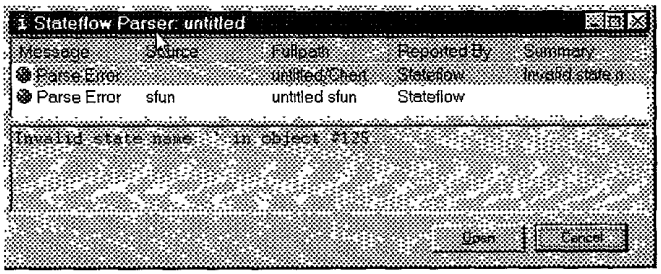
In this example, there are two states with the name A. Edit the Stateflow diagram and label the duplicate state with the text B.

The Stateflow diagram should look similar to this.



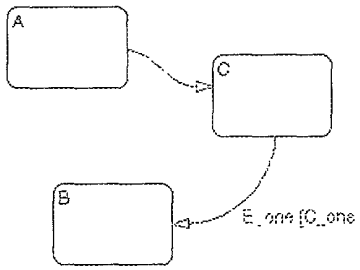
3 Reparse.

Choose **Parse Diagram** from the graphics editor **Tools** menu. This message is displayed in the pop-up menu and the MATLAB command window.



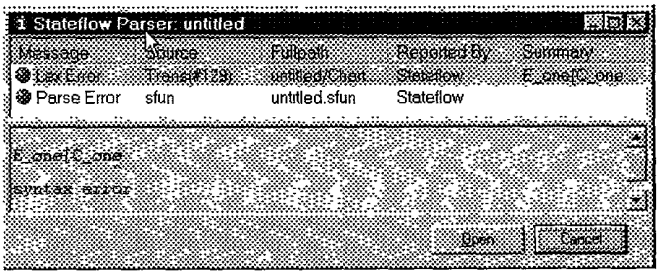
4 Fix the parse error.

In this example, the state with the question mark needs to be labeled with at least a state name. Edit the Stateflow diagram and label the state with the text `C`. The Stateflow diagram should look similar to this.



5 Reparse.

Choose **Parse Diagram** from the graphics editor **Tools** menu. This message is displayed in the pop-up window and the MATLAB command window.

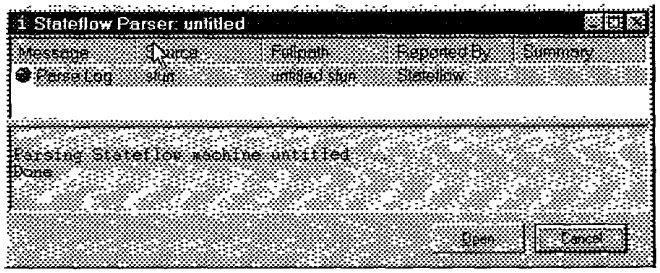


6 Fix the parse error.

In this example, the transition label contains a syntax error. The closing bracket of the condition is missing. Edit the Stateflow diagram and add the closing bracket so that the label is `E_one [C_one]`.

7 Reparse.

Choose **Parse Diagram** from the graphics editor **Tools** menu. This message is displayed in the pop-up window and the MATLAB command window.



The Stateflow diagram has no parse errors.

Error Messages

When building a target, you may see error messages from any of the following sources: the parser, the code generator, or from external build tools (make utility, C compiler, linker). Stateflow displays errors in a dialog box and in the MATLAB command window. Double-clicking on a message in the error dialog zooms the Stateflow diagram to the object that caused the error.

Parser Error Messages

The Stateflow parser flags syntax errors in a state chart. For example, using a backward slash (\) instead of a forward slash (/) to separate the transition action from the condition action generates a general parse error message.

Typical parse error messages include:

- "Invalid state name xxx" or "Invalid event name yyy" or "Invalid data name zzz"

  A state, data, or event name contains a nonalphanumeric character other than underscore.

- "State name xxx is not unique in objects #yyy and #zzz"

  Two or more states at the same hierarchy level have the same name.

- "Invalid transition out of AND state xxx (#yy)"

  A transition originates from an AND (parallel) state.

- "Invalid intersection between states xxx and yyy"

  Neighboring state borders intersect. If the intersection is not apparent, consider the state to be a cornered rectangle instead of a rounded rectangle.

- "Junction #x is sourcing more than one unconditional transition"

  More than one unconditional transition originates from a connective junction.

- "Multiple history junctions in the same state #xxx"

  A state contains more than one history junction.

Code Generation Error Messages

Typical code generation error messages include:

- "Failed to create file: modelName_sfun.c"

  The code generator does not have permission to generate files in the current directory.

- "Another unconditional transition of higher priority shadows transition # xx"

  More than one unconditional inner, default, or outer transition originates from the same source.

- "Default transition cannot end on a state that is not a substate of the originating state."

  A transition path starting from a default transition segment in one state completes at a destination state that is not a substate of the original state.

- "Input data xxx on left hand side of an expression in yyy"

  A Stateflow expression assigns a value to an **Input from Simulink** data object. By definition, Stateflow cannot change the value of a Simulink input.

Compilation Error Messages

If compilation errors indicate the existence of undeclared identifiers, verify that variable expressions in state, condition, and transition actions are defined.

Consider, for example, an action language expression such as a=b+c. In addition to entering this expression in the Stateflow diagram, you must create data objects for a, b, and c using the Explorer. If the data objects are not defined, the parser assumes that these unknown variables are defined in the **Custom code** portion of the target (which is included at the beginning of the generated code). This is why the error messages are encountered at compile time and not at code generation time.

Integrating Custom and Generated Code

The MATLAB Digest article, "Integrating Custom C-Code Using Stateflow 2.0," explains in detail how to integrate code that you write with code generated by Stateflow. This article is available at `http://www.mathworks.com/company/digest/june99/stateflow/`.

This section provides additional information on integrating code that you create with code generated by Stateflow from a Stateflow model.

Invoking Graphical Functions

To call a graphical function from your custom code:

1. Create the graphical function at the root level of the chart that defines the function (see "Creating a Graphical Function" on page 3-34).

2. Export the function from the chart that defines the function (see "Exporting Graphical Functions" on page 3-39).

   This option implicitly forces the chart and function names to be preserved.

3. Include the generated header file `chart_name.h` at the top of your custom code, where `chart_name` is the name of the chart that contains the graphical function.

   The chart header file contains the prototypes for the graphical functions that the chart defines.

11

Function Reference

This chapter contains detailed descriptions of Stateflow functions.

These functions operate on the machine.

Functions	
`sfexit`	Closes all Stateflow diagrams, Simulink models containing Stateflow diagrams, and exits the Stateflow environment.
`sfnew`	Creates and displays a new Simulink model containing an empty Stateflow block.
`sfsave`	Saves the current machine and Simulink model.
`stateflow`	Opens the Stateflow model window. See `stateflow`.

This function operates on a Stateflow diagram.

Functions	
`sfprint`	Prints the visible portion of a Stateflow diagram.

This function is independent of models and Stateflow diagrams.

Functions	
`sfhelp`	Displays Stateflow online help in the MATLAB help browser.

Purpose

Create a Simulink model containing an empty Stateflow block.

Syntax

```
sfnew
sfnew modelname
```

Description

`sfnew` creates and displays an untitled Simulink model containing an empty Stateflow block.

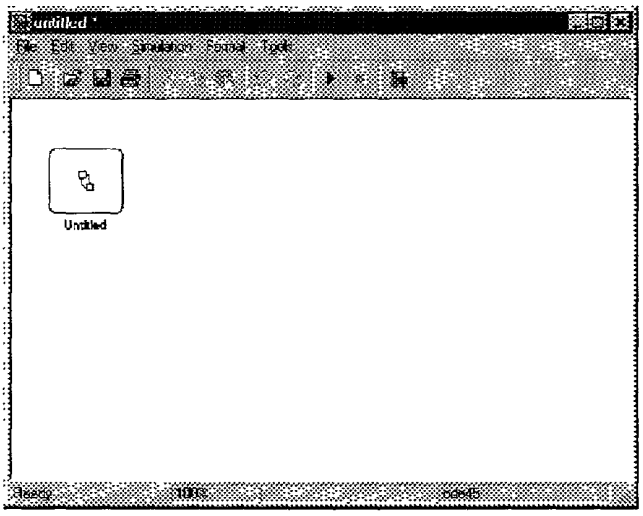
`sfnew modelname` creates a Simulink model with the title specified.

Example

Create an untitled Simulink model that contains an empty Stateflow block.

```
sfnew
```

The new model appears.



sfexit

Purpose

Close all Simulink models containing Stateflow diagrams and exit the Stateflow environment.

Syntax

```
sfexit
```

Purpose

Save a state machine and Simulink model.

Syntax

```
sfsave
sfsave ('machinename')
sfsave ('machine', 'saveasname')
sfsave ('defaults')
```

Description

`sfsave` saves the current machine and Simulink model.

`sfsave ('machinename')` saves the specified machine and its Simulink model.

`sfsave ('machine', 'saveasname')` saves the specified machine and its Simulink model with the specified name.

`sfsave ('defaults')` saves the current environement default settings in the defaults file.

stateflow

Purpose

Open the Stateflow model window.

Syntax

```
stateflow
```

Description

`stateflow` opens the Stateflow model window. The model contains an untitled Stateflow block, an Examples block, and a manual switch. The Stateflow block is a masked Simulink model and is equivalent to an empty, untitled Stateflow diagram. Use the Stateflow block to include a Stateflow diagram in a Simulink model.

Every Stateflow block has a corresponding S-function. This S-function is the agent Simulink interacts with for simulation and analysis.

The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow blocks into Simulink models, you can add event-driven behavior to Simulink simulations. You create models that represent both data and control flow by combining Stateflow blocks with the standard Simulink and toolbox blocksets. These combined models are simulated using Simulink.

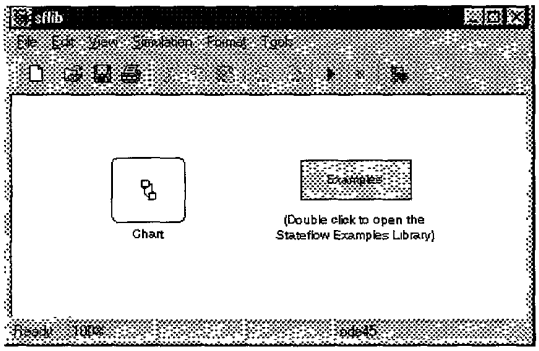
Example

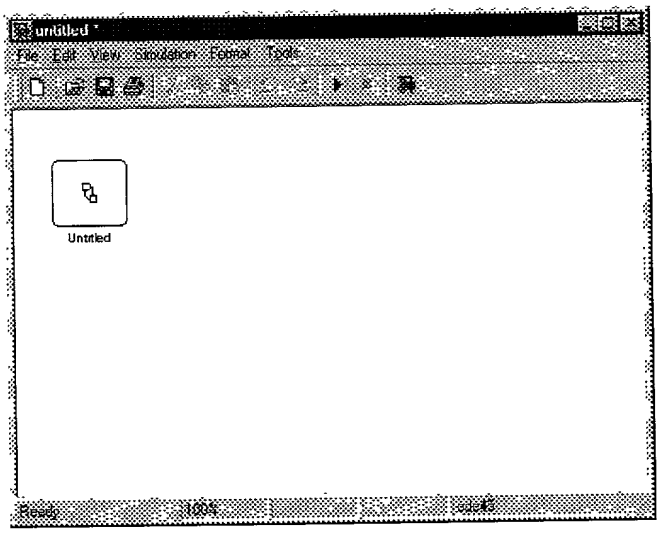
This example shows how to open the Stateflow model window and use a Stateflow block to create a Simulink model:

1 Invoke Stateflow.

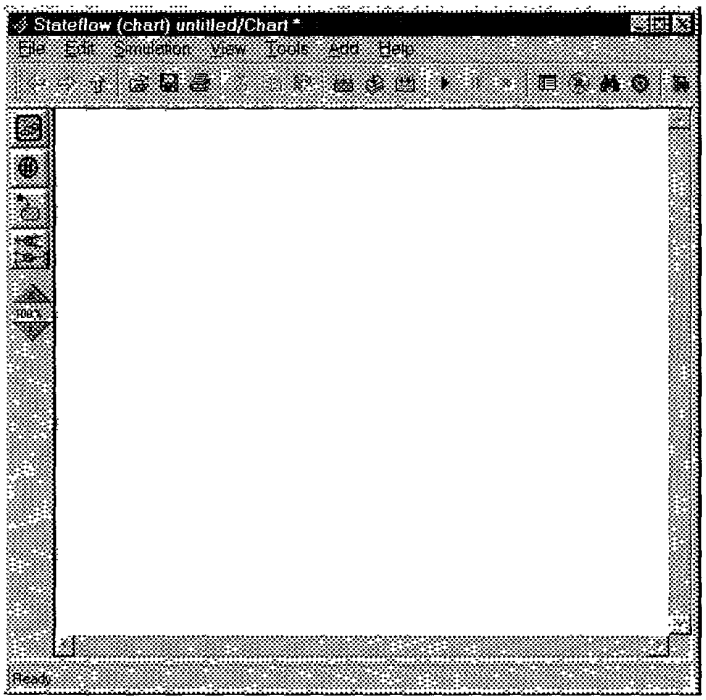
```
stateflow
```

The Stateflow model window and an untitled Simulink model containing a Stateflow block are displayed.





2 Double-click on the untitled Stateflow block in the untitled Simulink model to invoke a Stateflow editor window.



3 Create the underlying Stateflow diagram.

sfprint

Purpose

Print the visible portion of a Stateflow diagram.

Syntax

```
sfprint
sfprint ('diagram_name','ps')
sfprint ('diagram_name','psc')
sfprint ('diagram_name','tif')
sfprint ('diagram_name','clipboard')
```

Description

`sfprint` prints the visible portion of the current Stateflow diagram. A read-only preview window appears while the print operation is in progress. An informational box appears indicating the printing operation is starting.

See "Printing the Current View" on page 3-55, for information on printing Stateflow diagrams that are larger than the editor display area.

`sfprint ('diagram_name','ps')` prints the visible portion of the specified Stateflow diagram to a postscript file.

`sfprint ('diagram_name','psc')` prints the visible portion of the specified Stateflow diagram to a color postscript file.

`sfprint ('diagram_name','tif')` prints the visible portion of the specified Stateflow diagram to a `.tif` file.

`sfprint ('diagram_name','clipboard')` prints the visible portion of the specified Stateflow diagram to a clipboard bitmap (PC version only).

sfhelp

Purpose Display Stateflow online help.

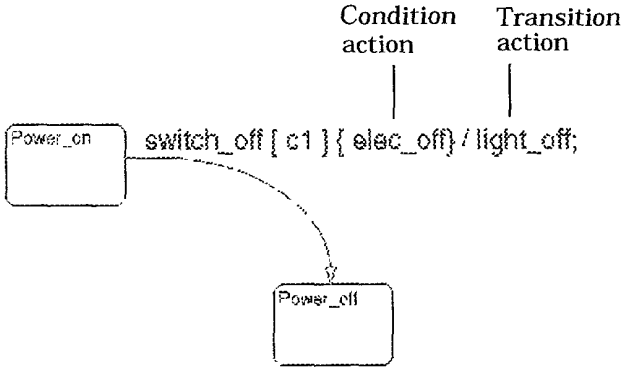
Syntax `sfhelp`

A

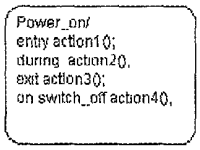
Glossary

Actions

Actions take place as part of Stateflow diagram execution. The action can be executed as part of a transition from one state to another, or depending on the activity status of a state. Transitions can have condition actions and transition actions. For example,



States can have `entry`, `during`, `exit`, and, `on` *`event_name`* actions. For example,



If you enter the name and backslash followed directly by an action or actions (without the `entry` keyword), the action(s) are interpreted as `entry` action(s). This shorthand is useful if you are only specifying `entry` actions.

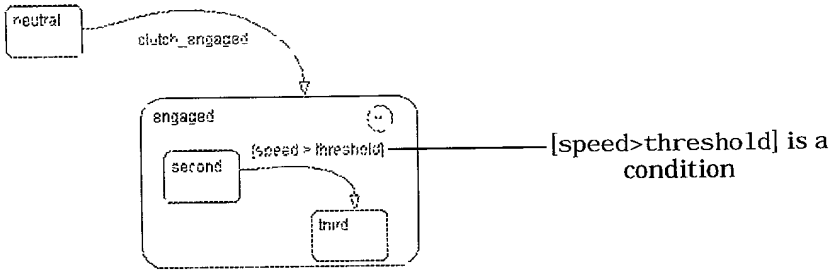
The *action language* defines the categories of actions you can specify and their associated notations. An action can be a function call, an event to be broadcast, a variable to be assigned a value, etc. For more information, see the section titled "Action Language" on page 7-37.

Chart Instance

A *chart instance* is a link from a Stateflow model to a chart stored in a Simulink library. A chart in a library can have many chart instances. Updating the chart in the library automatically updates all the instances of that chart.

Condition

A *condition* is a Boolean expression to specify that a transition occurs given that the specified expression is true. For example,

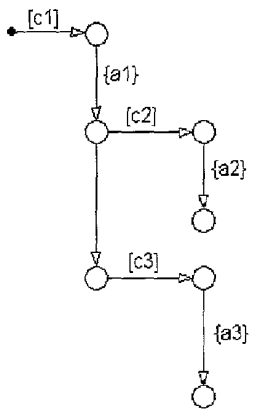


The action language defines the notation to define conditions associated with transitions. See the section titled "Action Language" on page 7-37 for more information.

Connective Junction

Connective junctions are decision points in the system. A connective junction is a graphical object that simplifies Stateflow diagram representations and facilitates generation of efficient code. Connective junctions provide alternative ways to represent desired system behavior.

This example shows how connective junctions (displayed as small circles) are used to represent the flow of an `if` code structure.




```
if [c1]{
     a1
     if [c2]{
        a2
     }else if [c3]{
       a3
     }
}
```

Name	Button Icon	Description
Connective junction		One use of a Connective junction is to handle situations where transitions out of one state into two or more states are taken based on the same event but guarded by different conditions.

See the section titled "Connective Junctions" on page 7-28 for more information.

Data

Data objects store numerical values for reference in the Stateflow diagram.

See "Defining Data" on page 4-13 for more information on representing data objects.

Data Dictionary

The *data dictionary* is a database where Stateflow diagram information is stored. When you create Stateflow diagram objects, the information about

those objects is stored in the data dictionary once you save the Stateflow diagram.

Debugger

See "Stateflow Debugger" on page A-11.

Decomposition

A state has a *decomposition* when it consists of one or more substates. A Stateflow diagram that contains at least one state also has decomposition. Representing hierarchy necessitates some rules around how states can be grouped in the hierarchy. A superstate has either parallel (AND) or exclusive (OR) decomposition. All substates at a particular level in the hierarchy must be of the same decomposition.

Parallel (AND) State Decomposition. *Parallel (AND) state decomposition* is indicated when states have dashed borders. This representation is appropriate if all states at that same level in the hierarchy are active at the same time. The activity within parallel states is essentially independent.

Exclusive (OR) State Decomposition. *Exclusive (OR) state decomposition* is represented by states with solid borders. Exclusive (OR) decomposition is used to describe system modes that are mutually exclusive. Only one state, at the same level in the hierarchy, can be active at a time.

Default Transition

Default transitions are primarily used to specify which exclusive (OR) state is to be entered when there is ambiguity among two or more neighboring exclusive (OR) states. For example, default transitions specify which substate of a superstate with exclusive (OR) decomposition the system enters by default in the absence of any other information. Default transitions are also used to specify that a junction should be entered by default. A default transition is represented by selecting the default transition object from the toolbar and then

dropping it to attach to a destination object. The default transition object is a transition with a destination but no source object.

Name	Button Icon	Description
Default transition		Use a Default transition to indicate, when entering this level in the hierarchy, which state becomes active by default.

See the section titled "Default Transitions" on page 7-21 for more information.

Events

Events drive the Stateflow diagram execution. All events that affect the Stateflow diagram must be defined. The occurrence of an event causes the status of the states in the Stateflow diagram to be evaluated. The broadcast of an event can trigger a transition to occur and/or can trigger an action to be executed. Events are broadcast in a top-down manner starting from the event's parent in the hierarchy.

Events are added, removed and edited through the Stateflow Explorer. See the section titled "Defining Events" on page 4-2 for more information.

Explorer

See "Stateflow Explorer" on page A-11.

Finder

See "Stateflow Finder" on page A-12.

Finite State Machine

A *finite state machine* (FSM) is a representation of an event-driven system. FSMs are also used to describe reactive systems. In an event-driven or reactive system, the system transitions from one mode or state, to another prescribed mode or state, provided that the condition defining the change is true.

Flow Graph

A *flow graph* is the set of flow paths that start from a transition segment that, in turn, starts from a state or a default transition segment.

Flow Path

A *flow path* is an ordered sequence of transition segments and junctions where each succeeding segment starts on the junction that terminated the previous segment.

Flow Subgraph

A *flow subgraph* is the set of flow paths that that start on the same transition segment.

Graphical Function

A *graphical function* is a function whose logic is defined by a flow graph. See "Working with Graphical Functions" on page 3-34.

Hierarchy

Hierarchy enables you to organize complex systems by placing states within other higher-level states. A hierarchical design usually reduces the number of transitions and produces neat, more manageable diagrams. See the section titled "Hierarchy" on page 2-11 for more information.

History Junction

A *History junction* provides the means to specify the destination substate of a transition based on historical information. If a superstate has a History junction, the transition to the destination substate is defined to be the substate that was most recently visited. The History junction applies to the level of the hierarchy in which it appears.

Name	Button Icon	Description
History junction		Use a History junction to indicate, when entering this level in the hierarchy, that the last state that was active becomes the next state to be active.

See these sections for more information:

- "History Junctions" on page 7-35

- "Example: Default Transition and a History Junction" on page 8-20
- "Example: Labeled Default Transitions" on page 8-21
- "Example: Inner Transition to a History Junction" on page 8-29

Inner Transitions

An *inner transition* is a transition that does not exit the source state. Inner transitions are most powerful when defined for superstates with XOR decomposition. Use of inner transitions can greatly simplify a Stateflow diagram.

See the sections titled "What Is an Inner Transition?" on page 7-24 and "Example: Inner Transition to a History Junction" on page 8-29 for more information.

Library Link

A *library link* is a link to a chart that is stored in a library model in a Simulink block library.

Library Model

A Stateflow *library model* is a Stateflow model that is stored in a Simulink library. You can include charts from a library in your model by copying them. When you copy a chart from a library into your model, Stateflow does not physically include the chart in your model. Instead, it creates a link to the library chart. You can create multiple links to a single chart. Each link is called a *chart instance*. When you include a chart from a library in your model, you also include its state machine. Thus, a Stateflow model that includes links to library charts has multiple state machines. When Stateflow simulates a model that includes charts from a library model, it includes all charts from the library model even if there are links to only some of its models. However, when Stateflow generates a stand-alone or RTW target, it includes only those charts for which there are links. A model that includes links to a library model can be simulated only if all charts in the library model are free of parse and compile errors.

Machine

A *machine* is the collection of all Stateflow blocks defined by a Simulink model exclusive of chart instances (library links). If a model includes any library

links, it also includes the state machines defined by the models from wich the links originate.

Notation

A *notation* defines a set of objects and the rules that govern the relationships between those objects. Stateflow notation provides a common language to communicate the design information conveyed by a Stateflow diagram.

Stateflow notation consists of:

- A set of graphical objects
- A set of nongraphical text-based objects
- Defined relationships between those objects

Parallelism

A system with *parallelism* can have two or more states that can be active at the same time. The activity of parallel states is essentially independent. Parallelism is represented with a parallel (AND) state decomposition.

See the section titled "State Decomposition" on page 7-7 for more information.

Real-Time Workshop

The Real-Time Workshop is an automatic C language code generator for Simulink. It produces C code directly from Simulink block diagram models and automatically builds programs that can be run in real-time in a variety of environments.

See the *Real-Time Workshop User's Guide* for more information.

RTW Target

An RTW target is an executable built from code generated by the Real-Time Workshop. See Chapter 9, "Building Targets" for more information.

S-Function

When using Simulink together with Stateflow for simulation, Stateflow generates an *S-function* (MEX-file) for each Stateflow machine to support model simulation. This generated code is a simulation target and is called the sfun target within Stateflow.

For more information, see *Using Simulink*.

Semantics

Semantics describe how the notation is interpreted and implemented behind the scenes. A completed Stateflow diagram communicates how the system will behave. A Stateflow diagram contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during Stateflow diagram execution.

Simulink

Simulink is a software package for modeling, simulating, and analyzing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates.

It allows you to represent systems as block diagrams that you build using your mouse to connect blocks and your keyboard to edit block parameters. Stateflow is part of this environment. The Stateflow block is a masked Simulink model. Stateflow builds an S-function that corresponds to each Stateflow machine. This S-function is the agent Simulink interacts with for simulation and analysis.

The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow diagrams into Simulink models, you can add event-driven behavior to Simulink simulations. You create models that represent both data and control flow by combining Stateflow blocks with the standard Simulink blocksets. These combined models are simulated using Simulink.

The *Using Simulink* document describes how to work with Simulink. It explains how to manipulate Simulink blocks, access block parameters, and connect blocks to build models. It also provides reference descriptions of each block in the standard Simulink libraries.

State

A *state* describes a mode of a reactive system. A reactive system has many possible states. States in a Stateflow diagram represent these modes. The activity or inactivity of the states dynamically changes based on transitions among events and conditions.

Every state has hierarchy. In a Stateflow diagram consisting of a single state, that state's parent is the Stateflow diagram itself. A state also has history that applies to its level of hierarchy in the Stateflow diagram. States can have

actions that are executed in a sequence based upon action type. The action types are: `entry`, `during`, `exit`, or on *event_name* actions.

Name	Button Icon	Description
State		Use a state to depict a mode of the system.

Stateflow Block

The *Stateflow block* is a masked Simulink model and is equivalent to an empty, untitled Stateflow diagram. Use the Stateflow block to include a Stateflow diagram in a Simulink model.

The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow blocks into Simulink models, you can add complex event-driven behavior to Simulink simulations. You create models that represent both data and control flow by combining Stateflow blocks with the standard Simulink and toolbox block libraries. These combined models are simulated using Simulink.

Stateflow Debugger

Use the *Stateflow Debugger* to debug and animate your Stateflow diagrams. Each state in the Stateflow diagram simulation is evaluated for overall code coverage. This coverage analysis is done automatically when the target is compiled and built with the debug options. The Debugger can also be used to perform dynamic checking. The Debugger operates on the Stateflow machine.

Stateflow Diagram

Using Stateflow, you create Stateflow diagrams. A *Stateflow diagram* is also a graphical representation of a finite state machine where *states* and *transitions* form the basic building blocks of the system. See the section titled "Anatomy of a Model and Machine" on page 2-4 for more information on Stateflow diagrams.

Stateflow Explorer

Use the *Explorer* to add, remove, and modify data, event, and target objects. See, "Exploring Charts" on page 6-3 for more information.

Stateflow Finder

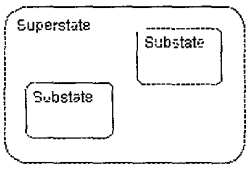
Use the *Finder* to display a list of objects based on search criteria you specify. You can directly access the properties dialog box of any object in the search output display by clicking on that object. See "Searching Charts" on page 6-8 for more information.

Subchart

A *subchart* is a chart contained by another chart. See "Working with Graphical Functions" on page 3-34.

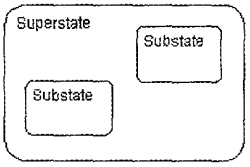
Substate

A state is a *substate* if it is contained by a superstate.



Superstate

A state is a *superstate* if it contains other states, called substates.



Supertransition

A *supertransition* is a transition between objects residing in different subcharts. See "Working with Supertransitions" on page 3-48 for more information.

Target

A *target* is an executable program built from code generated by Stateflow or the Real-Time Workshop. See Chapter 9, "Building Targets" for more information.

Topdown Processing

Topdown processing refers to the way in which Stateflow processes states and events. In particular, Stateflow processes superstates before states. Stateflow processes a state only if its superstate is activated first.

Transition

A *transition* describes the circumstances under which the system moves from one state to another. Either end of a transition can be attached to a source and a destination object. The *source* is where the transition begins and the *destination* is where the transition ends. It is often the occurrence of some event that causes a transition to take place.

Transition Path

A *transition path* is a flow path that starts and ends on a state.

Transition Segment

A *transition segment* is a single directed edge on a Stateflow diagram. Transition segments are sometimes loosely referred to as transitions.

Virtual Scrollbar

A *virtual scrollbar* enables you to set a value by scrolling through a list of choices. When you move the mouse over a menu item with a virtual scrollbar, the cursor changes to a line with a double arrowhead. Virtual scrollbars are either vertical or horizontal. The direction is indicated by the positioning of the arrowheads. Drag the mouse either horizontally or vertically to change the value.

See the section titled "Exploring Objects in the Editor Window" on page 3-12 for more information.